# Efficient SMT-based verification of software programs

Even Mendoza, Karine

*Awarding institution:*
King's College London


# Efficient SMT-Based Verification of Software Programs

By
Karine Even Mendoza

A thesis submitted in partial fulfilment for the
degree of Doctor of Philosophy

in the
Department of Informatics
School of Natural & Mathematical Sciences
King's College London

Submitted January 2020

**Keywords:** Formal Verification, Software Model Checking, Incremental Verification, Symbolic Algorithms, Satisfiability Modulo Theories (SMT), Bounded Model Checking (BMC), Function Summaries, Abstraction Refinement, Lattices.

# Abstract

In this thesis, we present efficient techniques for *satisfiability modulo theories-based model checking* (SMT-based MC) of software where the model is too large or complicated to analyse; real-world software once represented as a mathematical model faces the danger of the state-explosion problem where the size of the model grows exponentially, thus analysing the whole model becomes a challenge.

The SMT reasoning framework is one of the most successful approaches nowadays to deal with the state explosion problem when commonly combining additional techniques like symbolic algorithms, bounded model checking (BSMC), incremental modelling and reasoning, abstraction and counterexample-guided abstraction refinement (CEGAR). These approaches construct a model of the software with its specifications as a first-order formula while expressing domain-specific knowledge with first-order theories, thus creating smaller and simple models than with propositional logic modelling. The simplicity of the models improves the performance of the verification process and allows for reusing the model analysis for other tasks. However, finding a model that is sufficiently high-level to prevent reasoning from becoming prohibitively expensive but expressive enough to capture the software behaviour required for correctness, is a non-trivial task.

We describe novel SMT-based MC approaches in which a model of a software system is automatically analysed using these techniques to verify if the model satisfies its specifications or to find a real counterexample. The verification process is incremental through SMT summaries based on the structure of the program. The summaries are either Craig interpolants of previous successful verification tasks or user-defined. To avoid spurious counterexamples, each of the approaches introduces a refinement technique that deals with the over-approximative nature of modelling software in SMT framework. The *LB-CEGAR* algorithm uses lattices for efficient representation of library functions and gradual refinement of this representation; the *CEGAR-based theory refinement* algorithm uses the partial order according to precision between SMT theories to gradually refine program statements required for proving correctness; and the *function summarization modulo theories* algorithm uses summaries between different theories when verifying a software with many requirements.

We evaluate our approaches on benchmarks in C taken from SV-COMP (software verification competition), the robotics community, unimib (the University of Milano-Bicocca benchmarks), FunFrog tool benchmarks (the University of Lugano benchmarks), and our own. Our experimental results demonstrate that we can verify instances that existing model checking approaches failed to verify.

# Acknowledgements

I would like to express my sincere gratitude and countless thanks to my supervisors, Dr Hana Chockler and Professor Luca Viganò, for their continuous support and help during my PhD study. I further would like to thank my first supervisor, Dr Hana Chockler, for providing opportunities to promote me and my work, for her guidance and advice, enthusiastic encouragement, and useful and insightful conversations during my PhD and throughout the writing of this dissertation. I could not have imagined having a better advisor and mentor for my PhD study.

My grateful thanks are also extended to Prof. Natasha Sharygina, who gave me the opportunity to join and collaborate on the research at Università della Svizzera italiana (USI), as well as for her guidance and advice during my PhD studies.

I would also like to thank Dr Antti E. J. Hyvärinen for his guidance and advice in SMT and model checking, his exceptional contribution to the main results of my thesis, and for the helpful and insightful discussions during my PhD research. I thank Assistant Professor Grigory Fedyukovich for helpful discussions and the guidance and help he gave me in understanding the model checkers and incremental approaches. I thank Dr Aviel Even for the help and instructions with statistical software.

I thank the rest of the group at USI and KCL, Leonardo Alt, Sepideh Asadi, Martin Blicha, Matteo Marescotti, and Parvin Sadigova, for stimulating discussions, for the sleepless nights before deadlines, and for all the fun we have had in the last four years.

Further thanks go to the Department of Informatics for providing me with NMS faculty studentship (NMSFS) and to the Department of Informatics administration for their countless help and professional support during my studies.

Last but not least, I would like to thank my family: my parents, my wife, my daughter, my brothers, and my mother-in-law, for their enormous support and encouragement throughout my study, throughout writing this thesis and in my life in general.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Abbreviations

| | |
|---|---|
| **API** | Application Program Interface |
| **BDD** | Binary Decision Diagram |
| **BMC** | Bounded Model Checking |
| **BV** | A fixed-size Bit-Vectors |
| **BVP** | Bit-Vectors for Programs |
| **CEGAR** | CounterExample-Guided Abstraction Refinement |
| **CEX** | CounterExample |
| **CNF** | Conjunctive Normal Form |
| **DFS** | Depth-First Search |
| **DNF** | Disjunctive Normal Form |
| **DPLL** | Davis-Putnam-Logemann-Loveland algorithm |
| **EUF** | The logic of Equality and Uninterpreted Functions |
| **GCD** | Greatest Common Divisor |
| **LIA** | The logic of Linear Integer Arithmetic |
| **LIS** | Labelled Interpolation System |
| **LCM** | Least Common Multiple |
| **LRA** | The logic of Linear Real Arithmetic |
| **LTL** | Linear Temporal Logic |
| **LTS** | Labelled Transition System |
| **NRA** | The logic of Non-linear Real Arithmetic |
| **OM** | Out of Memory |
| **POSET** | Partially Ordered Set |
| **QF** | Quantifier-free Formula |
| **QFF** | Quantifier-free First-order Formula |
| **ROS** | Robot Operating System |
| **SAT** | Boolean Satisfiability |
| **SEM** | Standard Error Mean |
| **SMT** | Satisfiability Modulo Theories |
| **SSA** | Single Static Assignment |
| **TO** | Timed Out |
| **UDS** | User-Defined function Summary |
| **UFLIA** | Theory combination of EUF with LIA |
| **UFLRA** | Theory combination of EUF with LRA |
| **UFP** | Uninterpreted Functions for Programs |

**USSA** Unwound Single Static Assignment

# Chapter 1

# Introduction

Software nowadays is part of every aspect of our lives, from small personal devices to large industrial machinery and vehicles; now more than ever, there is a need to deal with errors, failures, faults, and vulnerabilities of software systems [All07, And11, Var15, Nes18, Boe19, Pat19, Top19]. Software is put through hours of testing to improve it to the point of acceptable risk for practical reasons. However, assuring a piece of software satisfies its requirements is still a complex and resource-consuming task. In this thesis, we explore ways to perform this non-trivial yet important task via software model checking framework, efficiently.

Model checking techniques [EC80, CE82, QS82, CES86, CGP99] verify a system against its requirements automatically and have been successfully applied for verification of protocols, controllers, and embedded software, already for several decades (e.g. [CLM91, CGH$^+$93, BVWW09, JR11, LVB$^+$12, GDH14, WDF$^+$15, Bon16, FBZ$^+$18]). The model checking framework, given a specification (safety properties), constructs a finite-state model of the system and checks if the model satisfies its specifications or finds a real counterexample. Software model checking often requires additional techniques to deal with performance and the represen-

tation of a non-finite state model of the task in hand. The performance and representation issues are resulting from two known problems: the *state explosion problem* in which the size of the model grows exponentially, and the *halting problem* that determines the termination of a software program and an input.

The *satisfiability modulo theories* (SMT) [DNS05] reasoning framework for model checking [BCCZ99, BCC+99, AMP06, GG06, AMP09] extends this capability by using a high-level description of the model and is one of the most successful approaches today to deal with the state explosion problem in model checking. This approach was presented originally with Boolean decision procedures and a bounded model checker (BMC) [BCCZ99, BCC+99, BCC+03] for symbolic model checking [BCM+92, McM93a, BCCZ99, BCC+99] as a scalable verification approach for larger systems, and in general, follows the same principle; the bounded model (that is, all loops and recursion calls are unwound to a given bound) is encoded and conjoined with the negation of a safety property to a first-order formula. The solver checks the satisfiability of this formula, and if there is no satisfying assignment, then no valid counterexample exists in the given bound, and the property holds in this bound. Since this approach constructs only finite formulas, it can deal with both problems, the state explosion problem and the halting problem. The effectiveness of SMT-based approaches in addressing the state-explosion problem depends on the verification problem and the theory in use.

A complementary task of the verification task in model checking is generating a counterexample when a property does not hold in the model; in software model checking, a counterexample is an execution of the program which violated this property. However, as we do model checking of the high-level description of the

system, a counterexample can be found in the model but does not necessarily exist in the system, in which case, we found a spurious counterexample.

A common way to deal with spurious counterexamples in the model is by refinement [CGJ+00, CCK+02, CGJ+03, CKSY05, GS05, KKNP09, RNO14]. The refinement replaces the high-level description of the model (part or all of it) with a more detailed description to avoid spurious counterexamples. The challenge is finding the balance between a high-level description of the problem and the required details for proving safety. One of the most successful refinement techniques in model checking is the counterexample-guided abstraction refinement (CEGAR) [CGJ+00, CGJ+03]: an automatic technique for refinement, which is guided by spurious counterexamples iteratively and has already been integrated into SMT-based model checking frameworks [HWZ08, Arm09, BW12, CNR13, LMN15, CGI+17b, CGI+18b].

In this thesis, I present four different CEGAR-based techniques, each of which uses additional techniques for constructing simpler models for efficient verification of the software with its properties via symbolic bounded model checking. We do incremental verification by using SMT function summaries to speed-up the current verification task and deal better with the state explosion problem. For each safety property, we construct a different model with the required details for this property only. We use function summaries of already verified code to avoid verification of the same part of the code twice; the SMT function summaries are Craig interpolants [Cra57] or user-defined [AAC+17] and can be more general and less architecture-specific. The initial modelling is done using Equality and Uninterpreted Functions (EUF) or extensions of EUF as uninterpreted functions and Linear Integer Arithmetic (UFLIA) and Uninterpreted Functions and Linear Real

Arithmetic (UFLRA). Thus, we construct a higher-level description by modelling with EUF than with propositional logic even if both are essentially the same.

These refinement techniques deal with the problem occurs when modelling software with EUF and SMT function summaries where the initial model is too simple and insufficient to prove the correctness of a property or find a real counterexample. We present four different refinement techniques to deal with each of the following cases.

1. The *summary refinement* algorithm handles the over-approximation nature of SMT function summaries.

2. The *LB-CEGAR-BMC* algorithm and the *LB-CEGAR* algorithm refine algebraic and mathematical functions (represented in the code as library functions).

3. The *counterexample-guided theory refinement* algorithm deals with local refinement of operators at the bit-vector level.

4. Last, the *theory-aware summary refinement* algorithm allows performing incremental verification with SMT function summaries of different theories.

The combination of these techniques helps to make reasoning efficient (time and space), the outputs (summaries and error trace) readable, and the model small and generalised while still being able to prove the correctness of the code by applying different refinement techniques if the model is too general. The general flow of the tool is sketched in Fig. 1.1. In this sketch, the model checking verifies each property separately via the assertion traversal component iteratively. Initially, the model checker encodes the software and a safety property in EUF with summaries (that is, the high-level model). Only if refinement is required, the model checker constructs a refined model (via the refiner engine and the encoder).

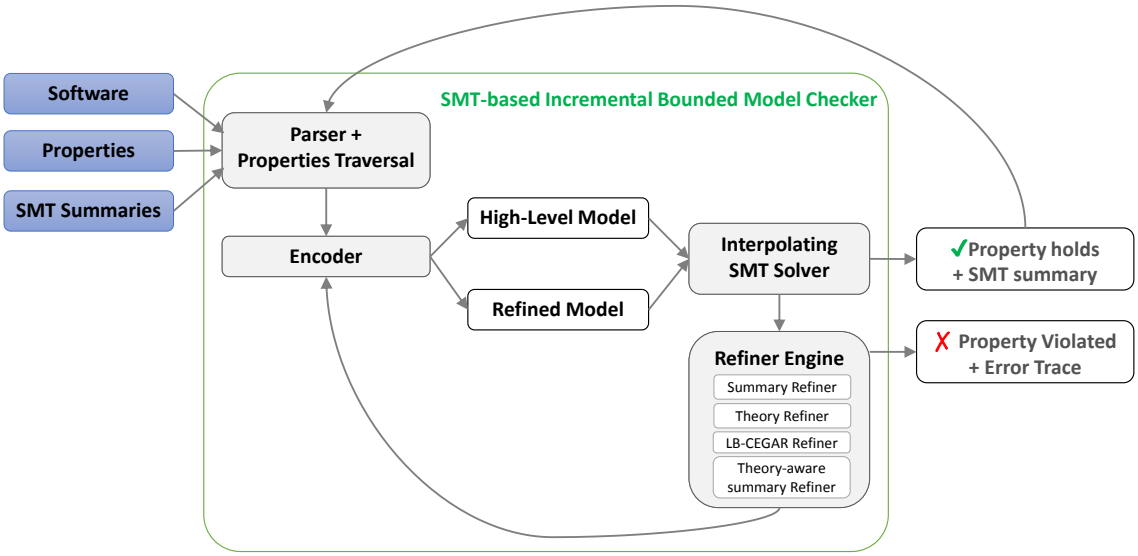


**Figure 1.1:** SMT-based incremental BMC with refinement (general flow).

We developed a novel SMT-based bounded model checker tool HiFrog for high-level incremental verification of software with refinement. The open-source tool HiFrog is available at `https://scm.ti-edu.ch/projects/hifrog/` [Git19b].

## 1.1 Contributions of the Thesis

The main contributions of this thesis are as follows.

- SMT-based incremental model checker tool called HiFrog[1] (the first SMT-based model checker that applied incremental verification with SMT summaries, Chap. 4).

[1] HiFrog is an open-source SMT-based model checker under the ownership of Università della Svizzera italiana (USI). The author of this thesis was part of the team that initially designed and developed the tool.

- Abstraction refinement techniques for efficient SMT-based incremental model checking to handle each of the following cases:
  - The over-approximation nature of SMT function summaries (Summary Refinement, Chap. 4).
  - Algebraic and mathematical functions in code (LB-CEGAR with lattices of guarded literals, Chap. 5 and Chap. 6).
  - Operators at a bit-vector level (Theory refinement, Chap. 7).
  - Efficient use of different theories in incremental verification with SMT summaries (Theory-aware summary refinement, Chap. 8).
- Evaluation of efficiency per refinement technique on sets of non-trivial benchmarks (our experimental results demonstrate that we can verify instances that existing model checking approaches failed to, Chap. 4-8).

## 1.2 Thesis Statement

In this thesis, the research direction hypothesis proposes to model the verification problem in different logics to use of SMT speeds-up calculation for scaling the model checking applications to process more complex and larger projects. The main challenge in this research direction relies on the ability to find a model that is expressive enough to capture relevant software behaviour while being sufficiently high-level to prevent prohibitively expensive reasoning. Failing to find such a model, results in either reporting spurious counterexamples due to abstractions or leading to instances that are too large to solve efficiently. In this thesis, I suggest handling this problem with abstraction refinement, present several refinement techniques for different verification tasks and unifying the different refinement

approaches here into one CEGAR loop, which is mainly relevant to future work. I explore in this thesis the claim that:

> *Modelling the verification problem in different SMT logics while incorporating smart and novel refinement techniques into the model checking process, is capable of verifying complex and larger software while dealing efficiently with both possible outcomes: all properties hold and a property violation.*

To test and explore the above claim, we created a new SMT-based model checker, HiFrog. It uses the CPROVER framework and converts a C program into a symbolic execution of the unwound program to create an instance of the verification problem as a first-order formula. I was participating in the development of the HiFrog tool in the first 3 years of my PhD studies with a significant contribution[2].

This research was done in collaboration with the USI Formal Verification and Security group to achieve the best research results; I present in Chap. 5 and Chap. 6 the main contribution of this thesis, namely, the concept of lattice-based refinement with its extension to a general algorithm for library function refinement, the LB-CEGAR algorithm. In Chap. 7 and Chap. 8, I describe other contributions, the theory refinement and the theory-aware summary refinement, which resulted from the collaborative work with the team at USI.

## 1.3 Thesis Outline

The rest of the thesis is structured as follows.

[2]See the git repository report at `https://scm.ti-edu.ch/projects/hifrog/repository/statistics` for more details.

I provide the theoretical background for this thesis in Chap. 2 and Chap. 3. I describe and review model checking approaches for software, and provide additional details about symbolic model checking and examples of the translation of code into the static single assignment (SSA) [CFR+89]. I provide details of the modelling phase in the context of incremental verification with function summaries and the encoding into a first-order formula for the SMT solver. I specify the techniques and tools I am using during the solving phase. Last, I discuss abstraction refinement techniques and provide the theoretical foundation of the refinement techniques I present in this thesis.

I present the function-summarization-based bounded model checker HiFrog in Chap. 4. I introduce the tool architecture and its various features. The features include the support to different encoding precisions, user-defined summaries, slicing, assertion optimisation, interpolation for function summaries and incremental verification. I evaluate the tool performance against SAT-based bounded model checking.

I describe and evaluate the effect on the performance and the ability to prove correctness (or to find a counterexample) of different refinement techniques, in Chap. 4, Chap. 5, Chap. 6, Chap. 7 and Chap. 8. I also give details on the initial encoding via EUF or extensions of EUF that each technique uses.

In Chap. 4, I describe in general the summary refinement algorithm, which takes place during the SMT encoding with summaries stage, and the interaction of the summary refiner component with other components of HiFrog in the architecture of the tool. This refinement technique refers in general to any SMT encoding and not specifically to EUF; later I show how to use this technique (combined with another refinement technique) when initially encoding the summaries with EUF in chapter 8.

In Chap. 5, I suggest using a lattice with properties of a mathematical function in the context of bounded model checking to model library functions in programs with EUF or extensions of EUF. I formalise a refinement process of the initial high-level description of the function for different sub-domains of the input till modelling with the function's full definition based on the lattice structure. I propose an algorithm for refinement based on lattices traversals in BMC (the LB-CEGAR-BMC algorithm) and an algorithm for constructing a lattice of literals from a set of properties. In Chap. 6, I present the LB-CEGAR algorithm, a full and generalised algorithm of the LB-CEGAR-BMC algorithm. The algorithm represents a function efficiently via a lattice of literals (as first-order formulas) and gradually refines the current representation of this function according to the partial order of literals where I do not necessarily have or use the full definition of this function; I describe additional techniques and heuristics to deal efficiently with this scenario. I present an evaluation of the lattice-based refinement algorithms with several mathematical functions and with different SMT encodings.

In Chap. 7, I present the counterexample-guided theory refinement approach. I describe a CEGAR-based algorithm that uses the partial order according to precision between SMT theories to gradually refine statements of a program up to the bit-level precision. However, the algorithm refines only the statements required for proving the correctness of the model, guided by spurious counterexamples and the location of a statement in the code. I describe the notation for and formalise the communication between two SMT theories in the context of software model checking. I present several heuristics with an evaluation of the effect of the refinement order of statements in the code on the time and memory performance of the algorithm.

In Chap. 8, I present the theory-aware summary refinement approach. The

approach allows for using function summaries between different theories for fully-automated incremental verification of software with many requirements. I present an algorithm that combines summary refinement with theory-aware refinement over an incremental verification framework, and I formalize the conversion that is carried out between different SMT encodings of the same function summary. To demonstrate the efficiency of the automated SMT-based incremental model checking framework, I provide a comparison with our semi-automated model checker (the initial version of HiFrog, Chap. 4) and CBMC model checker.

Last, I describe future work and conclude in Chap. 9. The last chapter also contains the list of publications used to compile this thesis.

# Chapter 2

# Preliminaries

In this chapter, we provide all required formalisms for the techniques we describe in chapters 4, 5, 6, 7, and 8. This chapter covers several topics: (i) lattices and sub-posets definitions and examples, (ii) modelling software for verification with a discussion regarding approximations and abstractions as part of the modelling process, (iii) *satisfiability modulo theories* (SMT) problem description with its usage and the required logical foundation, (iv) model checking in software with its general architecture and additional techniques for efficiency as *symbolic model checking*, *bounded model checking* (BMC), and *SMT-based model checking*, (v) further techniques that required their own section as *incremental verification*, *function summaries* and *abstraction refinement*. Topics (ii)-(v) are required for understanding this thesis in general, while topic (i) is required only for understanding Chap. 5 and Chap. 6.

## 2.1 Lattices

### 2.1.1 Partially ordered sets

A partially ordered set (***poset***) is a set with a binary relation where some pairs of elements in the set have a certain order, thus an element may precede another. However, not every pair of elements in the set are comparable.

Two elements in a poset $X$ are ***comparable*** if either $eR_be'$ or $e'R_be$ holds for a poset $\langle X, R_b\rangle$ and $e, e' \in X$ where $X$ is a set of elements and $R_b$ is a binary relation.

**Example 1.** Given a set of items $\{1, 2, 3\}$, the set of all its subsets (its powerset) $S = \{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$ and the binary relation $\subseteq$ is the poset $\langle S, \subseteq\rangle$ where only some of the pairs of elements are comparable, for example, elements $\{3\}$ and $\{1, 2\}$ are not comparable, and elements $\{1, 3\}$ and $\{1, 2, 3\}$ are comparable.

A poset is a ***chain*** if every two elements in the poset are comparable.

**Example 2.** The set of real numbers $\mathbb{R}$ and the binary relation $\leq$ is a chain (and also a totally ordered set).

A ***maximal element*** of a poset $\langle X, R_b\rangle$ is an element $e \in X$, thus

$$\neg\exists e' \in (X \setminus \{e\}).eR_be'.$$

**Example 3.** The maximal element of the poset $\langle S, \subseteq\rangle$, where $S = \{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$, is $\{1, 2, 3\}$ since there is no other element in the poset thus $e' \in (S \setminus \{\{1, 2, 3\}\})$ and $\{1, 2, 3\} \subseteq e'$.

For given sets $X_1, X_2$, a poset $\langle X_1, R_{b1}\rangle$ is a ***subposet*** of $\langle X_2, R_{b2}\rangle$ where $R_{b1}$ and $R_{b2}$ are binary relations, and $X_1 \subseteq X_2$, if

$$\forall e, e' \in X_1 . e R_{b1} e' \stackrel{\text{def}}{=} e R_{b2} e'.$$

An example of a subposet is illustrated in Fig. 2.1.

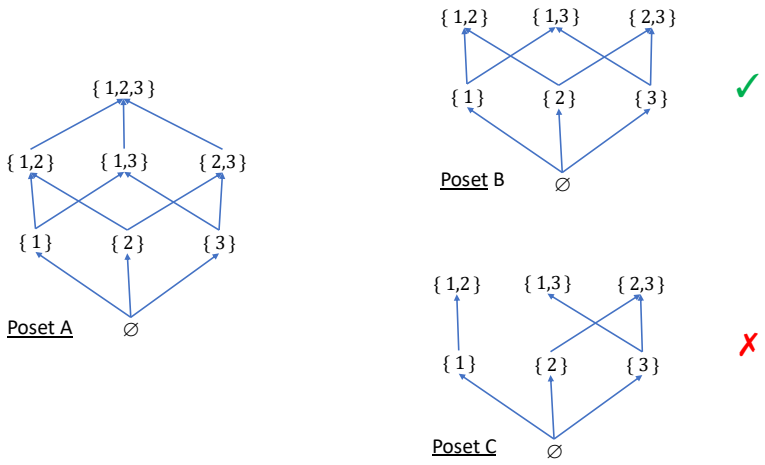


**Figure 2.1:** Poset A and two reduced posets: B and C. Only poset B is a subposet of poset A.

In Fig. 2.1, poset B (upper-right) is a subposet of poset A (left) since any two elements in the subposet maintain the same partial order as poset A (left). However, poset C (lower-right) is not a subposet of poset A (left), since for example, the element $\{1\}$ proceeds the element $\{1, 3\}$ in poset A (left) but does not proceed the element $\{1, 3\}$ in poset C (lower-right).

For a poset $\langle X, R_b\rangle$, let $Y \subseteq X$ and $l \in X$. Element $l$ is a ***lower bound*** of the set $Y$ if $\forall y \in Y . l R_b y$. The element $l$ is the ***greatest lower bound*** of $Y$ if $\forall e \in X . (\forall y \in Y . e R_b y) \implies e R_b l$. For a poset $\langle X, R_b\rangle$, let $Y \subseteq X$ and $u \in X$.

Element $u$ is a ***upper bound*** of the set $Y$ if $\forall y \in Y.yR_bu$. The element $u$ is the ***least upper bound*** of $Y$ if $\forall e \in X.(\forall y \in Y.yR_be) \implies uR_be$.

We define the *greatest lower bound* and the *least upper bound* as the **meet** ($\sqcap$) and **join** ($\sqcup$) operators on a poset $X$. We use these operators to define a lattice.

### 2.1.2 Lattices and semi-lattices

**Lattice.** A partially ordered set is a ***lattice*** if it has a meet ($\sqcap$) and a join ($\sqcup$) for any subset of its elements.

In Fig. 2.2, Lattice A is a lattice of all natural numbers ordered by $<$. Lattice B contains two elements $\{\bot, \top\}$ when $\top$ proceeds $\bot$. Lattice C is a lattice of all divisors of 30 ordered by divisors of each element, thus if $a$ divides $b$ then $a$ is a predecessor of $b$, where $\sqcap$ is the GCD and $\sqcup$ is LCM of a subset of Lattice C's elements.

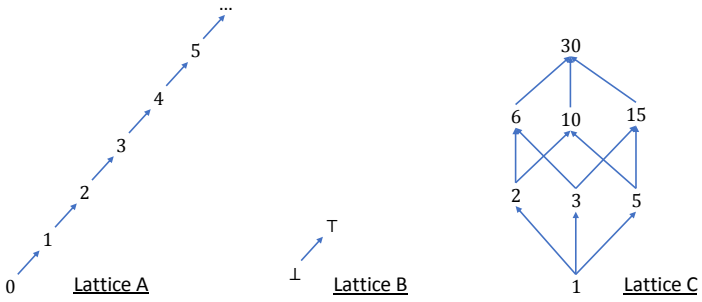


**Figure 2.2:** Examples of lattices.

**Finite lattice.** A **finite lattice** has a greatest element $\top$ and a least element $\bot$ thus every element in the lattice is between $\bot$ to $\top$. In Fig. 2.2, Lattice A is not a finite lattice. Lattice B and Lattice C are finite lattices.

**Semi-lattice.** A ***meet semi-lattice*** is a partially ordered set that has a $\sqcap$ for any subset of its elements (but not necessarily $\sqcup$). A ***join semi-lattice*** is a partially ordered set that has a $\sqcup$ for any subset of its elements (but not necessarily $\sqcap$). In Fig. 2.1, *poset B* is a meet semi-lattice; we present its $\sqcap$ and $\sqcup$ operators in the next paragraph when we discuss subset lattices. For the general definition of semi-lattices, which is not required for understand this thesis, we refer the reader to [Gar15], Chap. 5.

**Subset lattice.** For a given set $X$, the family of all subsets of $X$, partially ordered by the inclusion operator, forms a *subset lattice* $L(X)$. The $\sqcap$ and $\sqcup$ operators are defined on $L(X)$ as *intersection* and *union*, respectively. The top element $\top$ is the whole set $X$, and the bottom element $\bot$ is the empty set $\emptyset$. The height of the subset lattice $L(X)$ is $|X| + 1$, and all maximal chains have exactly $|X| + 1$ elements. We note that $L(X)$ is a De-Morgan lattice [Bir67], as meet and join distribute over each other. In Fig. 2.1, Poset A is a subset lattice as it is a lattice of all the subsets of $\{1, 2, 3\}$, while Poset B is its reduce meet semi-lattice.

In this thesis, we consider only lattices where $X$ is a finite set and only meet semi-lattices $S(X)$ that are reduced semi-lattices of $L(X)$ ordered by the inclusion operator. For general definitions, additional explanations and examples regarding lattices, we refer the interested reader to [Gar15].

## 2.2 Modelling Software, Abstractions and Approximations

A model of the program under verification aims to represent the relevant part of the software system for proving the correctness of its specification by abstracting irrelevant parts of the code. An approximation of the actual behaviours of the program is a common way of abstraction. An over-approximation of the program contains all the program's behaviours (and possibly other additional behaviours), while an under-approximation of the program contains only the program's behaviours (but not necessarily all of them).

In the context of software verification, a model which is an *over-approximation* of a program can include behaviours that violate the specification that do not exist in the actual program; however, because the model is an over-approximation of the actual code, it always includes all the behaviours of the program, including those which violate the specification[1]. Therefore, we say that if no behaviour that violates the specification was found in the model, then there is no such behaviour in the program. However, a model that is an under-approximation of the program includes some of the behaviours that violate the specification in the program (but the program's under-approximation can only include behaviours that exist in the actual program). Hence, any behaviour that violates the specification found in this model exists in the program, but we may require a refined model to detect all such behaviours.

In this thesis, we focus on modelling C code for bounded model checking, where all the behaviour of a program is all its possible executions. We use initially the

[1]In this thesis, we consider only safety properties as our specification. We describe the input and output of our approach in Sec. 2.4, *Bounded Model Checking* paragraph.

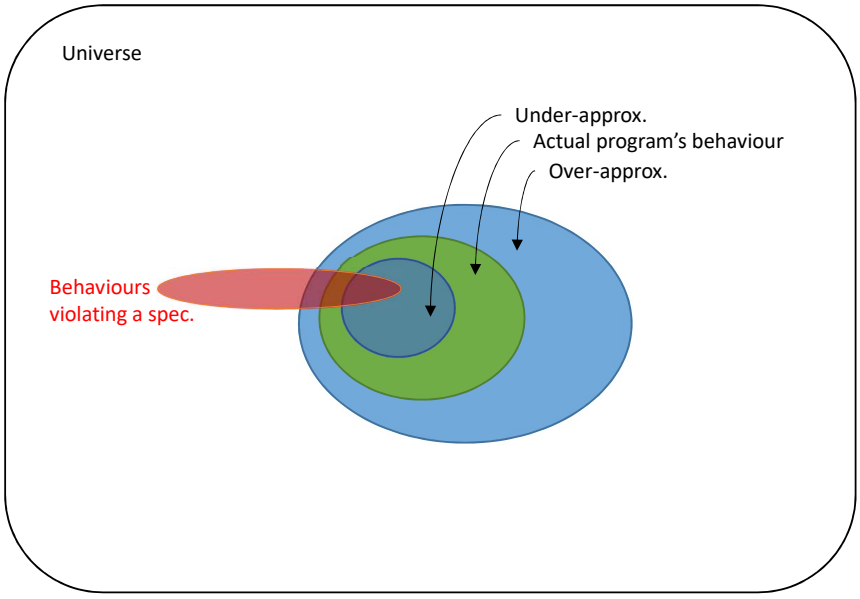


**Figure 2.3:** Models of a program for verification of a specification

Unwound Single Static Assignment (USSA) form while constructing an under-approximation of the number of recursive function calls and loop-iterations in the program (see Sec. 2.4), and then we use different SMT logics to construct a first-order formula that is an over-approximation of the loop-free program (see Sec. 2.3 and Sec. 2.4.2).

## 2.3 Satisfiability Modulo Theories

The *satisfiability modulo theories* (SMT) decision problem of formulas is a problem of determining whether a formula's variables have a satisfying assignment with respect to combinations of background theories expressed in first-order logic.

The SMT decision problem is an extension of the original *Boolean Satisfiability Problem* (SAT). The problem of determining whether a *propositional logic formula* (Boolean expression) is evaluated to true by finding an assignment to all

its Boolean variables (each of which is either true or false); this assignment is called a satisfying assignment, and the formula is satisfiable. This formula can contain Boolean variables, the constants true and false, and $\wedge, \vee$, and $\neg$ operators, and unlike first-order logic has no non-logical symbols nor quantifiers.

As the focus of this thesis is using first-order theories for software verification, we describe next the first-order logic that also contains the formal definitions required for propositional logic.

### 2.3.1 First-order logic

In this thesis, we express the verification problem in first-order logic; we briefly describe its syntax and semantics; for full definitions and examples see [Lin06]. The *syntax* is a collection of symbols that are valid in a formula in a first-order logic, and the *semantics* determine the formula interpretation in first-order logic.

A *formula* in general is a sequence of symbols: logical symbols and a signature's symbols. *Logical symbols* are the quantifiers $(\forall, \exists)$, logical connectives $(\wedge, \vee, \neg, \implies, \iff)$, punctuation symbols (brackets, comma, etc.), equality symbol $(=)$, and variables. A *signature*, $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{P}, ar)$, is a union of *non-logical symbols* of $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{P}$, which is a pairwise disjoint sets of constants $\mathcal{C}$ (functions with arity 0), function symbols $\mathcal{F}$ and predicate (relation) symbols $\mathcal{P}$, and the arity function, $ar$, that is defined for all functions and predicates (that is, $ar : \Sigma \to \mathbf{N}$).

To describe a first-order formula, we require the following rules;

**Sort.** A *sort* is a set of constants. For example, the Boolean sort $\mathbb{B} = \{\top, \bot\}$ consists of the Boolean constants, true and false.

Terms are defined inductively, where

**Function Mapping.** Given a set of sorts $\{T_{ret}, T_1, \ldots, T_n\}$, a *function op* : $T_1 \times \ldots \times T_n \rightarrow T_{ret}$ maps a (possibly empty) sequence of constants $v_1, \ldots, v_n$ such that for $1 \leq i \leq n$, $v_i \in T_i$ to a *return value* $v_{ret} \in T_{ret}$.

**Variables and Constants.** *variables* are $V = \{v_i | i \in \mathbf{N}\}$ and are not part of $\Sigma$. *Constants* are the symbols in $\mathcal{C}$ in $\Sigma$.

In this thesis, we use the definition below when describing the results in Chap. 7 to be consistent with OpenSMT2's definitions. Thus, functions mapping empty sequences are *variables* and are *constants* if the return value is fixed.

**Terms.** A set of *terms* $\mathcal{R}_\Sigma$ is defined inductively, thus a *term* is either a constant, a variable, or an application of a function $op(t_1, \ldots, t_n)$ where $t_i$ are, recursively, terms with a return value in the sort $T_i$ and $ar(op) = n$. These rules are captured by the following grammar:

$$
\begin{aligned}
term ::= & \; const \\
| & \; var \\
| & \; f(term, \ldots, term)
\end{aligned}
$$

where $const \in \mathcal{C}$ is a constant, $var$ is a variable, and $f \in \mathcal{F}$ is a function symbol with arity equal to the number of terms in parentheses.

**Equalities.** Given two terms $t$ and $t'$ with a value from sort $T$ and sort $T'$ respectively, an *equality between two terms* is $t = t'$.

**Atomic Formulas.** An *atomic formula* is either an equality between two terms or a predicate expression over terms $p(t_1, \ldots, t_m)$ where $p \in \mathcal{P}$, $t_i$ are, recursively, terms with a value from sort $T_i$, and $ar(p) = m$.

**Literals.** A *literal* is an atomic formula or its negation.

Note that, an "'atomic formula"' is as defined in the paragraph above, that is, equalities or predicates only.

Using the above we define a *first-order formula* inductively.

**Binary connectives.** Given first-order formulas $\phi$ and $\psi$, any of the following formulas are also a first-order formula (e.g.,): $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \implies \psi$, and $\phi \iff \psi$.

**Quantified Formula.** Given a first-order formula $\phi$ and a variable $x$, $\forall x\phi$ and $\exists x\phi$ are first-order formulas.

**First-order Formulas.** A *first-order formula* is either a literal, a formula of a binary logical connective between two first-order formulas, or a quantified formula.

**Quantifier-free Formulas.** In this thesis, we use only *quantifier-free first-order formula* (QFF); QFFs are either literals or formulas of a binary logical connective between two QFFs. Formally, a set of such *formulas* $\mathcal{S}_\Sigma$ is built inductively using the following grammar:

$$
\begin{aligned}
\mathit{formula} ::= {} & \mathit{Bvar} \\
& \mid p(\mathit{term}, \ldots, \mathit{term}) \\
& \mid \mathit{term} = \mathit{term} \mid \top \mid \bot \mid \neg \mathit{formula} \\
& \mid \mathit{formula} \wedge \mathit{formula} \mid \mathit{formula} \vee \mathit{formula}
\end{aligned}
$$

where *Bvar* is a Boolean variable, $p \in \mathcal{P}$ is a predicate symbol with arity equalling to the number of terms in parentheses, and $\top$ and $\bot$ are Boolean constants denoting *true* and *false* respectively.

In most cases in this thesis we use the usual infix notation together with parentheses to express the well-known arithmetic and logical functions.

**Free Variables.** Given a formula $\phi$ and a variable $x$, we say that $v$ is a *free variable* if there exists an occurrence of $v$ that is not bound by any quantifier in $\phi$.

**Example 4.** *For the formula* $\forall x.\exists y.x = a \wedge y! = b$ *with* $V = \{a, b, x, y\}$*, the variables* $a$ *and* $b$ *are free variables since both are not bound by* $\forall$ *nor* $\exists$ *in* $x = a \wedge y! = b$*.*

Formally we define $x$ to be a *free variable* in $\phi$ if $\phi$ is an atomic formula; or (closure) if $\phi$ is $\phi_1 \odot \phi_2$ ($\odot$ is any of the Binary connectives) and $x$ is free in either $\phi_1$ or $\phi_2$; if $\phi$ is $\neg\psi$ and $x$ is free in $\psi$; if $\phi$ is $\forall y.\psi$, $x$ is free in $\psi$ and $y$ is a different variable than $x$; and if $\phi$ is $\exists y.\psi$, $x$ is free in $\psi$ and $y$ is a different variable than $x$.

**Sentences.** A formula $\phi$ is a *sentence* if $\phi$ has no free variables.

Given a signature $\Sigma$, *first-order theories* and *quantifier-free first-order theories* are defined as follows;

**First-order Theories.** A *first-order theory* $\mathcal{T}$ of $\Sigma$ is a set of sentences in first-order logic consisting of symbols from $\Sigma$, thus, if $\mathcal{T} \models \phi$ then $\phi \in \mathcal{T}$, for sentences $\phi$ over $\Sigma$.

**Quantifier-free First-order Theories.** In this thesis, we use only QFF's hence, we also define *a quantifier-free first-order theory* $\mathcal{T}_\Sigma \subseteq \mathcal{S}_\Sigma$ as a set of formulas defined over the signature $\Sigma$. When $\mathcal{T}$ is clear from the context, we call a formula from $\mathcal{S}_\Sigma$ a *Satisfiability modulo theory (SMT) instance.* We model verification

problems in the next chapters with the quantifier-free SMT theories of equality and uninterpreted functions (EUF), fixed-size bit-vectors (BV), linear real arithmetic (LRA), linear integer arithmetic (LIA), and non-linear arithmetic over the real numbers (NRA), and with theory combination of EUF with LIA (UFLIA) and EUF with LRA (UFLRA).

To define the semantics meaning via interpretation of formulas and terms, we use the following definitions (relevant to QFFs, as we use only quantifier-free theories here).

**First-order Structures.** Given $\Sigma$, a *structure* $M = (D, I)$ is the domain of discourse $D$ that is a non-empty set and the interpretation $I$ of the set of constants in $\mathcal{C}$, (written as: $c_1^M, ..., c_k^M$), functions in $\mathcal{F}$, (written as: $f_1^M, ..., f_l^M$), and predicates in $\mathcal{P}$, (written as: $p_1^M, ..., p_m^M$) in $D$, where $X^M$ is the interpretation of $X$ (a constant, a function or a predicate) in $M$.

**Assignments.** A *variable assignment* $\mu_v$ associates each variable with an element in $D$, that is, a function $\mu_v : \{v_i | i \in \mathbf{N}\} \rightarrow D$. Then the assignment is extended to *assignment* $\mu$ to all terms in $\Sigma$, adding the closure of the evaluation of the set of functions with all terms that have been evaluated to elements in $D$ and constants.

**Formula Evaluation.** Each formula $\phi$ is evaluated to either true or false under $\mu$ in $M$. (atomic formula) If $\phi$ is an equality between two terms $t$ and $t'$, then $\phi$ is true if and only if $\mu(t) = \mu(t')$; or if $\phi$ is an $n-$place predicate $p(t_1, .., t_n)$, then $\phi$ is true if the evaluation $< ev_1, .., ev_n >$ of its terms $< t_1, .., t_n >$ is in its interpretation (that is, $< ev_1, .., ev_n > \in p^M$); and (closure) if $\phi$ is $\phi_1 \odot \phi_2$ ($\odot$ is any of the Binary connectives) or $\phi$ is $\neg\phi_1$, then $\phi$ is evaluated according to the truth table of each the logical connectives.

**Satisfiable Formula.** In general, if $\psi$ is evaluated to true under $M$, we say that $M$ *satisfies* $\psi$ and use the notation $M \models \psi$. A QFF $\psi$ is $\mathcal{T}-$satisfiable if and only if there exists a structure $M$ such as that $\mathcal{T} \cup \psi$ is satisfiable (**SAT**). However, $\mathcal{T} \cup \psi$ is unsatisfiable (**UNSAT**) if and only if there is no such structure $M$.

### 2.3.2 Craig interpolation

We extract a function summary from a solver's proof (**UNSAT** case) via *Craig interpolation* [Cra57]. We use Craig interpolation theorem to create an **interpolant** (a formula) from the solver's proof. An interpolant for a formula exists if the formula can be partitioned into two parts with common symbols in both partitions, and every non-logical symbol in the interpolant occurs in both of them. Each interpolant is described (with some changes) as a function summary in the verifier. These are reusable for other closely related verification tasks (Sec. 2.5) and over-approximation representations (Sec. 2.2) of part of the code (e.g., function's code). We describe function summaries with their usage in model checking in Sec. 2.5.1.

Given a pair of formulas $(A, B)$ where the formula $A \wedge B$ is **UNSAT**, the definition of an interpolant $I$ for a pair $(A, B)$ is as follows;

**Interpolant.** *Craig interpolant* of $(A, B)$ is a formula $I$ such that,

- $A \Rightarrow I$.
- $I \wedge B$ is still **UNSAT**.
- $Vars(I) \subseteq (Vars(A) \cap Vars(B))$ that is $I$ is defined over the common symbols of $A$ and $B$.

$Vars(F)$ is the set of symbols (variables) in formula $F$.

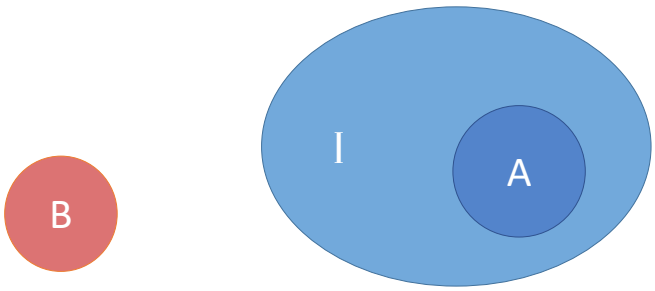


**Figure 2.4:** Interpolant $I$ of formula $A \wedge B$.

We describe the relations between $A, B$ and $I$ schematically in Fig. 2.4, thus every state reachable from $A$ is also reachable from $I$, and in that sense, we say that $I$ is an over-approximation of $A$.

In this thesis, we compute interpolants from resolution refutations via the labelled interpolation system (LIS) framework; we use the implementation in OpenSMT2 as is. A *refutation* is a proof that no counterexample is reachable in $k$ or fewer steps in a model, for some $k \in \mathbf{N}$. The **labelled interpolation system** (LIS) framework computes an interpolant $I$, given a refutation of $A \wedge B$ and a labelling function [Pud97, McM05, DKPW10, RAF+13].

The *labelling function* maps variables in a refutation's clauses to a set of labels $\{a, b, ab\}$. The labelling function maps an $A$'s local variable to $'a'$, a $B$'s local variable to $'b'$, and an $A$ and $B$'s shared variable to one of the items in $\{a, b, ab\}$. Different labelling functions have a different mapping of the shared variables. A *shared variable* occurs in $A$ and in $B$, and a *local variable* occurs in either one of them.

Interpolants of different *strength* and *size* can be computed for the pair $(A, B)$ by choice of the labelling function. Given two interpolants $I$ and $I'$ for the pair

$(A, B)$, if $I \Rightarrow I'$ then we say that the interpolant $I$ is stronger than an interpolant $I'$ and that the interpolant $I'$ is weaker than an interpolant $I$; Fig. 2.5 presents different strength of interpolants, where $I$ is a stronger interpolant than $I'$ and $I''$. The size of an interpolant depends on the structural size of the formula itself (e.g., number of equalities in EUF or the number of connectives in propositional logic).

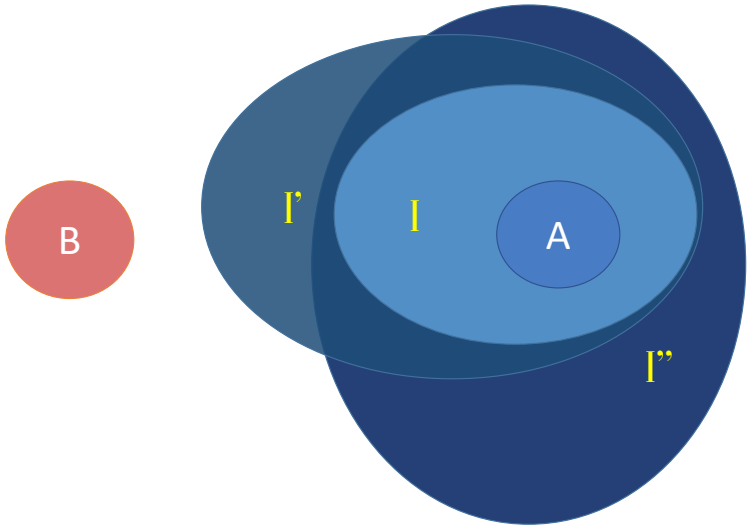


**Figure 2.5:** Different strength of interpolants ($I, I'$, and $I''$) of formula $A \wedge B$.

**Application.** In the context of this thesis, unsatisfiable formulas originate from bug-free programs, and thus the summaries express that no trace allowed by the function body leads to a violation of the considered safety specification. We define the loop-free instance of the program as $A$ and the violated property (an assert) as $B$, then from the **UNSAT** proof of $f = A \wedge B$, we construct interpolant $I$ via the labelled interpolation system (LIS) framework. We store the interpolant after compressing and generalizing it as a summary.

We describe in Sec. 4.2.5 the mechanism of obtaining summaries by interpolation in HiFrog.

### 2.3.3 Tools

SMT solvers are usually built on SAT solver (e.g., MiniSAT2 [ES04]) with the addition of theory solvers. A query to the solvers is given in the SMT-LIB language (using currently the SMT-LIB2 format [BST10]). The SMT-LIB language is the syntax of the SMT-LIB standard that is a common standard for SMT systems in general [Sat19]. The formula is parsed, simplified, translated into CNF form, and transformed from an SMT problem into a SAT formula (by replacing atomic formulas in Boolean variables). A formula is in conjunctive normal form (CNF) if and only if it is a conjunction of disjunctions of literals. Then the solver repeatedly tries to find a satisfying valuation for the SAT formula and check consistency under the domain-specific theory via its theory solver (usually via the DPLL(T) framework [GHN+04, NO05, NOT06] or via other lazy or eager approaches).

All results in this thesis require the OpenSMT2 SMT solver. For solving with theory combination, we use the Z3 SMT solver required for some of the algorithms described in Chap. 5 and Chap. 6.

**OpenSMT2.** We briefly describe the structure and logic of the OpenSMT2 SMT solver (see [HMAS16] for a detailed description). The solver supports reading an SMT-LIB2 file and interacting through an application-program-interface (API). The problem is converted into an SMT formula $\varphi$ and simplified into $\varphi_s$, both on the propositional level, for example flattening nested conjunctions and removing Boolean constraints, and on the theory level, where for instance asserted equalities are used to compute variable substitutions. The formula $\varphi_s$

is then translated into conjunctive normal form $\varphi_{CNF}$, which is provided as an input to the SAT solver. The SAT solver provides the theory solvers with assignments satisfying $\varphi_{CNF}$. If the assignment is discovered to be unsatisfiable by the theory solver, the theory solver returns a clause that prevents the SAT solver from producing similar inconsistent assignments. The standard SAT solving algorithm involves producing learned clauses and resolution guided by the conflict graph [SS99]. The process terminates when either $\varphi_{CNF}$ becomes unsatisfiable, or when the SAT solver finds a theory-consistent truth assignment. We query the solver only via its C/C++ binary API.

**Z3.** We briefly describe additional relevant details in the Z3 solver for this thesis, see [DMB08b] for a general and detailed description. The Z3 solver has a rich front-end support; in our implementation, we query the solver through either an SMT-LIB2 textual format or the C/C++ binary API. We do not store any information back from the Z3 solver for later use (e.g., function summaries via interpolation). The Z3 solver is a DPLL-based SAT solver and uses a model-based theory combination method to incrementally reconcile models maintained by each theory [dMB08a]. In the implementation of HiFrog, we query with theory combination of EUF with LIA and EUF with LRA, with a core theory solver to handle equalities and uninterpreted functions and a linear arithmetic solver (a satellite solver based on the algorithm used in Yices [DdM06]) to handle linear arithmetic.

### 2.3.4 Incremental solving

Incremental solving is a general approach in SAT and SMT solvers that allows altering the constraints in the solver via push and pop operations between two dif-

ferent checks for satisfiability. Incremental solving for verification is particularly useful during the refinement of an over-approximating formula of the verification problem when a relatively small part of the formula is changed between two different checks for satisfiability. In this thesis, we use a semi-incremental or a non-incremental solving mode in the OpenSMT2 SMT solver, and an incremental solving mode in the Z3 SMT solver.

## 2.4 Software Model Checking

One of the common and successful approaches for software verification is *model checking*. Model checking automatically and systematically explores a finite-state-model of software or hardware to determine whether the system meets all its requirements. The requirements are given as a specification, in this thesis, as a set of *safety properties*. As the focus of this thesis is verification of software, these properties express that none of the error states is reachable during the execution of the program. Examples of safety properties in software include buffer under- and over-flow, division by zero, index array out of bounds, NULL pointer dereferencing, memory leaks, and race-conditions. Model checkers, given a specification, can present a real counterexample when the model violates a safety property.

Software model checking often requires additional techniques to deal with performance and the representation of real-world problems when represented as a mathematical model. In this thesis, we represent the verification problem as a finite formula, via SMT-based symbolic bounded model checking techniques. We handle the state-explosion problem via abstraction and abstraction refinement techniques for scaling the model checking applications to process much complex and larger projects. The *state explosion problem* is the problem in which the size

of the model grows exponentially. We describe the general flow in Fig. 2.6 and dedicate the next sections to describing and defining each part in detail (Sec. 2.4.1 and Sec. 2.4.2); in Sec. 2.4.3, we describe the symbolic execution representation and give details regarding the symbolic compiler we use in HiFrog.

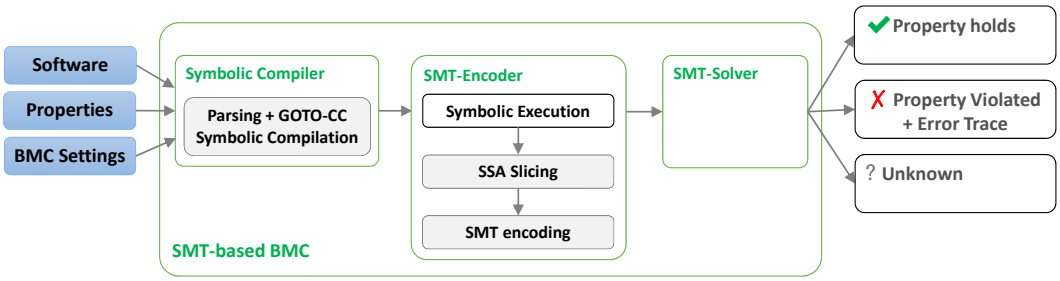


**Figure 2.6:** SMT-based symbolic bounded model checker (general architecture).

### 2.4.1 Symbolic model checking

The formalism of *symbolic model checking* was presented originally with *Binary Decision Diagrams* (BDDs) [BCM+92, McM93a] and later with SAT procedures [BCCZ99, BCC+99], for a compact modelling of the verification problem. SAT procedures with *bounded model checking* (BMC) for checking linear temporal logic (LTL) formulas deals with the state-explosion problem in BDD-based symbolic model checking by reducing the problem into propositional logic [BCCZ99].

We represent hardware or software system implicitly as a formula (e.g., in temporal logic or first-order logic) based on the labelled transition system (LTS) of the system represented as a Kripke structure [Kri59].

**Kripke Structure.** Let $AP$ be the set of atomic propositions. A Kripke structure $M = (S, I, R, L)$ represents the transition system, where

- $S$ is a finite set of the reachable states in the system.
- $I$ is a subset of states in $S$ that are initial states.
- $R$ is a set of state transitions between states in $S$, thus $R \subseteq S \times S$.
- $L$ is a labelling function thus $L : S \to 2^{AP}$.

Note that, (i) the set of atomic propositions $AP$ represents properties in the system; (ii) the labelling function $L$ maps the set of atomic propositions that are true in $s$, for each $s \in S$; (iii) we represent the set of atomic propositions as a Boolean function $L : S \to 2^{AP}$, where $S$ are states in $M$ and $2^{AP}$ is a Boolean string of $0s$ and $1s$; and (iv) a *path* in $M$ is an infinite sequence of states $\pi = s_0 s_1, s_2, \ldots$ thus $s_0 \in I$, $s_i \in S$ (for $i = 0, 1, 2, \ldots$), and $R(s_i, s_{i+1})$ holds.

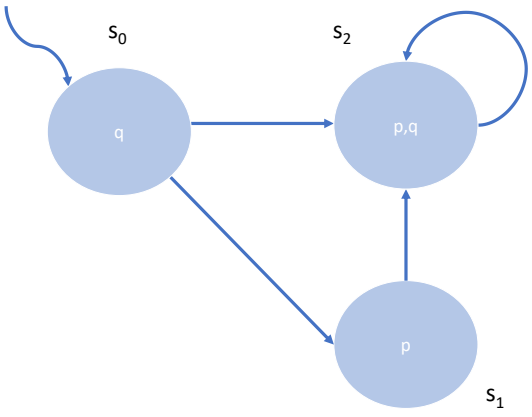


**Figure 2.7:** A sketch to describe schematically the Kripke structure in Ex. 5.

**Example 5.** Fig. 2.7 describes an example of $M = (S, I, R, L)$ with $AP = \{p, q\}$ and $I = \{s_0\}$, $S = \{s_0, s_1, s_2\}$, $R = \{(s_0, s_1), (s_0, s_2), (s_1, s_2), (s_2, s_2)\}$, and $L$ maps $s_0$ to "01", $s_1$ to "10", and $s_2$ to "11".

In BDD-based symbolic model checking, we use temporal logic to model the formula representing the system; in SAT-based symbolic model checking, we use propositional logic with bounded model checking semantics instead. We define the bounded model checking problem and discuss the reduction from BDD-based symbolic model checking to SAT-based symbolic model checking in the next paragraph; in Sec. 2.4.2 we describe SMT-based symbolic model checking, based on SAT-based symbolic model checking. See Sec. 2.3.1 for the definition of first-order logic; propositional logic is included in first-order logic but excludes non-logical symbols, predicates about non-logical symbols, and quantifiers.

**Bounded Model Checking.** In *Bounded Model Checking* (BMC), we convert a program $\overline{P}$ to a *loop-free* program $P$ by unwinding all loops and recursive calls up to a given bound. The approach constructs a model that is an under-approximation of the behaviour of the program $\overline{P}$. It is an under-approximation, as bound model checking can only verify that the specification holds in the given bound and not in general. In this thesis, we use the bounded model checking to verify safety properties (assert statements) in $C$ programs.

The bounded model checking problem is defined as follows; let $P$ be a *loop-free* program, $k$ a predefined bound ($k \in \mathbf{N}$), and $t$ a *safety property*. We represent $P$ as a transition system $M$. The bounded model checking problem amounts to determining whether all states of $P$, reachable within a predefined bound $k$, satisfy $t$.

The BMC problem is determined by solving the BMC formula, which is the problem's encoding as a formula to a SAT solver. The *BMC formula* in proposi-

tional logic with a Boolean encoding of the states in $S$ is

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_{i-1}, s_i) \wedge (\neg t). \tag{2.1}$$

The first part of the formula in Eq. 2.1 represents the set of all execution of length $k$ in $P$. The second part of the formula in Eq. 2.1 is the negation of the safety property, $t$. The reduction from Kripke structure for LTL formula to SAT-based symbolic BMC is described in [BCCZ99], where LTL formula in bounded model checking represents all possible path up to depth $k$, that is, all $k$-length prefixes.

A *satisfying assignment* of the BMC formula (from a SAT solver) is a concrete example of a $k$-length execution path that violates $t$ at some point. The bounded model checker terminates once finds a counterexample, that is, a bounded execution of $P$ that falsifies $t$, or proves the absence of such executions within the bound, that is, there is no satisfying assignment. Bounded model checkers commonly repeat the checking with a larger $k$ ($k \in \mathbf{N}$) until either finding a real counterexample or proving the specification holds in general. In our tool HiFrog, this part can only be done manually.

The current implementation of HiFrog allows controlling the value of $k$ as an input only and offers no such automatic iterative deepening depth-first search (DFS). We define the transition system for function summarization bounded model checking in Sec. 2.5.1; the basic definition of the bounded model checking problem stays the same. Next, we describe symbolic model checking with SMT reasoning (instead of SAT).

### 2.4.2 SMT-based symbolic model checking

The SMT-based model checking approach verifies the first-order formula and is successful in verifying software in a scalable way. In this approach, the model checker encodes all bounded executions of $P$ as an SMT formula, conjoins it with the negation of $t$ as before (Eq. 2.1), and invokes an SMT solver to check the satisfiability of the resulting formula. We describe $P$, a *loop-free* program and $t$ a *safety property* in first-order logic. That is, $P$ is represented as a quantifier-free first-order formula and $t$ is a first-order formula over the variables of $P$. The description of the state transition system $M$ does not require now a Boolean encoding of the states in $S$[2], and $AP$ is a set of literals based on predicates in first-order logic.

In case that a first-order formula is deemed unsatisfiable, the program is safe, that is, $P$ satisfies $t$. Otherwise, a satisfying assignment found by the SMT solver is used to build a concrete counterexample. Depending on the theory used by the SMT solver, an abstract counterexample can also be *spurious*, that is, not corresponding to any concrete execution. This situation arises when the theory is too abstract, and hence the resulting over-approximation of the behaviours of the program is too coarse as discussed in Sec. 2.2. In this case, the program is re-verified with a different theory, a combination of theories or additional literals.

### 2.4.3 The static single assignment form

The Static Single Assignment (SSA) form [CFR$^+$89, CFR$^+$91] is a way to represent code such that each variable is assigned exactly once. If a variable is declared without any assignment, we refer its value as a non-deterministic value according

[2]We can represent the states and the initial states in the system as quantifier-free first-order formulas.

to its data-type. If a variable has been assigned more than once in the code, we create several (enumerated) instances of the variable. For example, if an integer variable 'a' first set as '5' and then (later on) as '6', we represent it as two different assignments each is with a different enumerated instance of 'a': `a#0 = 5;` and `a#1 = 6;`. The symbol # states which instance of $x$ it is in HiFrog. Other tools can have a different symbol to represent the instance number.

In HiFrog, we use the USSA (Unwound Static Single Assignment) approximation by converting the symbolic execution of the unwound program (up to some bound) to its SSA representation. The symbolic execution is an intermediate *goto-program*, where all the loops and recursion calls are unwound to the pre-determined number of iterations. The goto-program is created via the Goto-CC symbolic compiler provided by the CPROVER framework [CKL04, cpr19] and is an input usually to other verification tools as HiFrog. The Goto-CC symbolic compiler [got19] supports the ANSI-C language (as it is in *gcc* compiler).

```
1   int main()
2   {
3       int a;
4       while (1)
5       {
6         assert (a!=5);
7         a -= 5;
8       }
9   }
10
```

**Figure 2.8:** Code example of a simple C program.

To demonstrate the different stages of encoding in HiFrog, we use a simple C code in Fig. 2.8, with `-unwind` 4. The GOTO program in Fig. 2.9 and the

USSA form in Fig. 2.10, are both tree-like expressions in HiFrog[3], the printed representation is merely for giving a general idea about each stage encoding.

```
main /* main */
        signed int a;
     1: IF !(1 != 0) THEN GOTO 2
        ASSERT a != 5
        a = a - 5;
        GOTO 1
     2: dead a;
        return NONDET(signed int);
        END_FUNCTION
```

**Figure 2.9:** The GOTO program of the code in Fig. 2.8.

```
/* Partition 1 */
    |main::1::a!0#2| = |main::1::a!0#1| - 5
    |main::1::a!0#3| = |main::1::a!0#2| - 5
    |main::1::a!0#4| = |main::1::a!0#3| - 5

  ASSERT (or
            (|main::1::a!0#1| != 5)
            (|main::1::a!0#2| != 5)
            (|main::1::a!0#3| != 5)
            (|main::1::a!0#4| != 5))
```

**Figure 2.10:** The SSA representation of the code in Fig. 2.8 with USSA approximation with unwind parameter equal to 4.

HiFrog prints a GOTO program representation by using the `-show-program` option and the SSA representation in SMT-LIB2 [BST10] style by setting the compilation flag `DEBUG_SSA_PRINT` to true.

## 2.5 Incremental Verification

In this thesis, we use incremental verification approach to verify different components in the system in an incremental manner, wherein each cycle we prove the

[3]Since we first unwind the loops and then perform the function calls analysis (callee-caller).

system correctness for a subset of components. Incremental verification allows us to find a violation of the specification faster and to consume fewer resources by re-verifying each cycle only part of the system.

We represent the specification as a set of safety properties $t_1, ..., t_n$. In each cycle, we verify a single property and re-verify only the necessary components for this property. That is, we construct from the symbolic execution a first-order formula with the components required for proving the correctness of a safety property $t_i$ (a single property from the set of properties). We replace any pre-visited components with its over-approximation description, that is, with its function summary. We verify each of the formulas for all properties, $t_1, ..., t_n$, and avoid re-verification of components required for more than one property by using function summaries. We describe function summaries in the next section.

Note that, we use the mechanism of function summary for incremental verification in the thesis. Other techniques can be used instead of function summaries. We discuss additioanl techniques in the context of incremental verification in Chap. 3.

We use function summaries with the USSA approximation as a mean of incremental verification in HiFrog, where each call of a recursion-free function is a unique component in the system, and each assert statement is a single property. Incremental verification in HiFrog includes the following steps:

- The specification of the software system is represented as a set of safety properties, $\{t_1, ..., t_n\}$.
- We represent each function call with its function description as a component in the system. We refer to each component as a *partition*.
- During each cycle, per safety property $t_i \in \{t_1, ..., t_n\}$, we

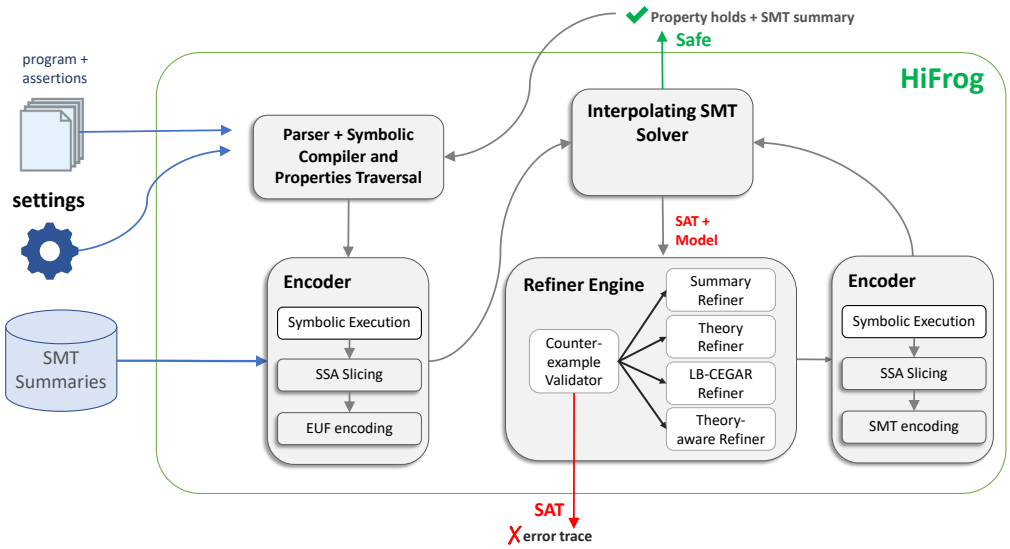


**Figure 2.11:** SMT-based symbolic incremental bounded model checking architecture used in HiFrog.

– Update the representation of the software system to contain only the required components for verifying $t_i$ (Fig. 2.11, Encoder). We apply SSA slicing on the original symbolic execution to keep only the currently required components. SSA slicing retains only the variables in the SSA representation with syntactic dependence on the variables in $t_i$.

– Check which components have a function summary (for pre-visited components) and whether these components are affected. That is, whether the function summary of each of these components is still valid in the sliced SSA representation when trying to verify if $t_i$ holds in $P$. The summary refiner in HiFrog discards any invalid function summary and replaces it with the original description (Fig. 2.11, Refiner Engine). It is then necessary to re-verify each of the affected components.

– Verify new components and re-verify affected comportments with respect to $t_i$. We avoid re-verification of unaffected components by using their valid function summaries (Fig. 2.11, Interpolating SMT Solver).

The steps above describe the main cycle in Fig. 2.11, that is, parser-encoder-solver for each of the properties $t_i \in \{t_1, ..., t_n\}$. The inner cycle encoder-solver-refiner presents another aspect of incrementally in HiFrog, where we incrementally and gradually adjust the representation of the current verification problem (i.e., verifying whether $t_i$ holds in $P$) via a counterexample-guided abstraction refinement (CEGAR) framework. We discuss the inner cycle in more details in Sec. 2.6.

### 2.5.1 Function summaries

*Function summarization* is a generic technique for abstracting programs. A *function summary* describes a component in the system. In software, it is usually a function or a function call in the code. The summary is a formula on the input and output conditions of the function that is reused and replaces the analysis of pre-visited code to reduce time and space complexity of the code analysis.

In this thesis, we use function summaries as a mean of performing incremental verification or abstraction of known or common (in a specific system) functions. The verification task can be broken down as a sequence of smaller closely related tasks, to cope better with the state explosion problem and to speed-up the current verification task. The verification tasks are assert-statements in the code. The summaries can be extracted from an unsatisfiable SMT formula of a successful verification task and can be over-approximations of the actual behaviour of the functions. We can then use these extracted function summaries for other verifi-

cation tasks instead of analysing their code again. We describe this process in HiFrog with a refinement of over-approximated summaries in Chap. 4, Sec. 4.2.

**Modelling Programs with Function Summaries.** In HiFrog, we model programs in the context of (i) bounded model checking (we model only loop-free programs) and (ii) satisfiability modulo theories (we need to support any SMT theory) with functions summaries. It raises two major limitations: (a) the functions summaries generated with HiFrog cannot represent functions with infinite loops or recursions, and (b) we avoid defining new functions in SMT in order to support theories other than EUF and its extensions. That is, we can still use functions given as part of each theory, for example, it is still possible to use "+" between two integers in LIA, but we cannot introduce new functions to LIA.

A solution for (b) is to assume all functions have no arguments and use global variables instead, defined uniquely to each occurrence.

**Example 6.** *To understand better the limitation and solution of (b), we use the following code example.*

```
1   int foo(int input)
2   {
3     return (input < 0) ? 0 : input++;
4   }
5
6   int main()
7   {
8     int x;
9     int a=foo(x);
10    int b=foo(a);
11    assert(b > 0);
12  }
```

By using LIA, we prove the assert holds and thus, our function summary is in LIA. However, pushing a definition of function $foo$ to the solver, which is `(declare-fun |foo#0| (Int) Int)`, would only allow summaries expressed with a theory combination; in this example is EUF (to be able to define new functions) with LIA (to be able to express $<$ and $+$).

Our choice of implementation does not include theory combination.[4] Instead, we define *input_*#1 and *input_*#2 as global variables because we have two calls to function $foo$. We set *input_*#1 to be $x$ and *input_*#2 to be $a$ (after the assignment on line 9).

We define function summaries in the context of SMT-based BMC as follows; a *loop-free program* is a tuple $P = (F, main)$, such that $F$ is a finite set of non-recursive functions, and $main \in F$ is an entry point. Let set $\hat{F}$ gather all function calls from $F$, where $\hat{f}$ is a call of function $f$. In $\hat{F}$ we distinguish different calls to the same function $f$ by enumerating them as $\hat{f}_1, \ldots, \hat{f}_n$. The set of state transitions in the system is a subset of $\hat{F} \times \hat{F}$ and it models the function calls relations (caller-callee relation) in the loop-free program $P$.

A *summary* of a function $f$ is a relation over the input and output variables of $f$ that over-approximates the precise behaviour of $f$. That is, if a formula $f_{precise}$ encodes the body of $f$, and $f_{sum}$ encodes its summary, then $f_{precise} \implies f_{sum}$ must hold.

We give a high-level description of function summaries and usage in HiFrog in Chap. 4, Sec. 4.2. We formalize the description of function summaries and programs in more details in Chap. 8, Sec. 8.3.

[4]From my conversations with OpenSMT2's developers, it is clear that theory combination does not currently supported in OpenSMT2 and they were not sure whether this will be implemented in the near future.

### 2.5.2 User-defined function summaries

User-defined function summaries are encoded in SMT-LIB2 format for light-weight theories (e.g., EUF, LRA and LIA) and use the *easy-to-read by human* nature of SMT encoding to create high-level summaries to unsupported or complicated to describe functions. Several examples of these functions can be $isnan()$, $isinf()$ or even % in C, which have no straight-forward support in SMT. Other examples can contain more complicated functions, as trigonometric functions or `math.h` implementation of other common mathematical functions.

The user-defined function summary can be described once and used later many times as needed. The summaries shall be updated only if the definition or the implementation of the function changes or if a summary refinement within the theory (as we discuss in Chap. 4) or between theories (as we discuss in Chap. 8) is required.

User-defined summaries can contain the actual definition of the function or an over-approximation of it. For example, Fig. 2.12 is a user-defined summary of an over-approximation of the expression of a property of the trigonometric functions sin and cos: $\sin^2 x + \cos^2 x = 1$.

```
(define-fun |c::nonlin#0| (
  (|c::nonlin::x!0| Real)
  (|hifrog::?fun_start| Bool)
  (|hifrog::?fun_end| Bool)
  (|c::nonlin::?retval| Real) )  Bool
  (= 1 |c::nonlin::?retval|)
)
```

**Figure 2.12:** UDS for $\sin^2(x) + \cos^2(x)$ (over-approximated).

An example of a user-defined summary without approximation is shown in Fig. 2.13, and is a user-defined summary of an expression of a property of the trigonometric function cos: $\cos(-x) = \cos x$, where $|\_cos\#0|$ is a function.

```
(define-fun |cos_neg#0| (
  (|cos_neg::a| Int)
  (|hifrog::fun_start| Bool)
  (|hifrog::fun_end| Bool)
  (|cos_neg::?retval| Int) )  Bool
    (let ((?def274  |cos_neg::?retval|))
    (let ((?def275  (= (|_cos#0| |cos_neg::a|) ?def274)))
    (let ((?def276  (= (|_cos#0| (- |cos_neg::a|)) ?def274)))
    (let ((?def277  (and ?def275 ?def276)))
  ?def277
)))))
```

**Figure 2.13:** UDS for $\cos(-x) = \cos x$.

We describe the usage of user-defined summaries in HiFrog in Chap. 4, extend it to a library of summaries in Chap. 5, and further extend it to formulate a system with a set of guarded literals based on these libraries in Chap. 6. Some examples of user-defined summaries are available on HiFrog's webpages [HiF17b, HiF19].

## 2.6 Abstraction Refinement in Model Checking

Abstraction is one of the most common approaches today for achieving efficiency and better complexity in both modelling and solving of a verification task. The abstraction aims to keep in the model only the relevant parts for proving the correctness of a program and hence are capable of creating simpler and smaller models at the cost of loss of information. These techniques may construct an abstract description that is not expressive enough to determine the correctness of a program as we discuss in Sec. 2.2. We use refinement to deal with this problem.

### 2.6.1 Counterexample-guided abstraction refinement

One of the most common approaches for dealing with an approximative model is the Counterexample-Guided Abstraction Refinement (CEGAR) approach

[CGJ+00, CGJ+03]. The CEGAR framework begins with verifying an approximative model while iteratively refining it, until eliminating all spurious counterexamples results. During each refinement iteration, the algorithm checks the current model and performs a feasibility check of the current counterexample whenever a violation is found. An infeasible counterexample is called spurious counterexample and requires additional iterations of refinement; a feasible or a real counterexample is reported to the user, as shown in the CEGAR loop sketch in Fig. 2.14.

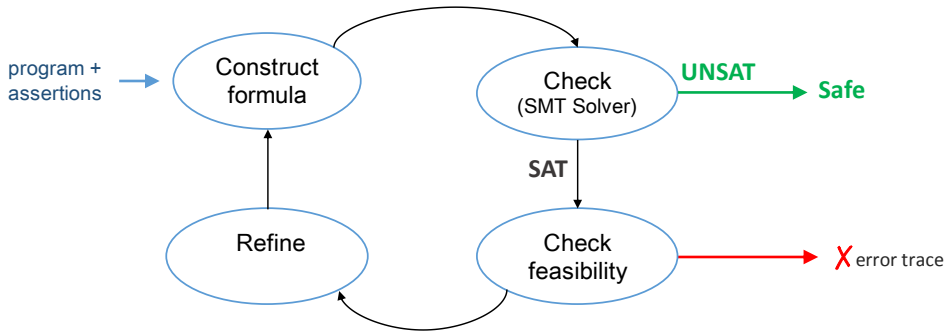


**Figure 2.14:** General CEGAR loop.

In the SMT-based model checking approach followed in this thesis, the CEGAR loop contains the following steps:

- **Construct** a first-order formula in EUF of the current verification problem, that is, a program $P$ with a safety property $t$ (Fig. 2.14, *construct a formula* stage during the first iteration of the refinement loop).
- **Verify** $t$ in $P$ via an SMT solver; the result is either **UNSAT** or **SAT** with a counterexample $CEX$[5] (Fig. 2.14, *check* stage).

[5]The result can also be *unknown*; in that case, we treat it as **SAT** result with a spurious counterexample and continue with the refinement.

- **Employ a feasibility check** of the current counterexample each time the solver reports a **SAT** result with a counterexample (Fig. 2.14, *check feasibility* stage); the *check* stage and the *check feasibility* stage can use a different solver and different SMT theories to complete their tasks.

- **Terminate the refinement** once the solver returns **UNSAT** (that is, $t$ holds in $P$) or once a counterexample passed the feasibility check (Fig. 2.14 reporting *SAFE* in green colour or *Error trace* in red colour). In the latter case, the CEGAR algorithm generates an error trace based on the feasible current counterexample.

- **Continue the refinement** as long as spurious counterexamples still exist (Fig. 2.14, *refine* stage) by constructing a refined first-order formula (Fig. 2.14, *construct a formula* stage).

In the next chapters, we apply several abstraction techniques; we do symbolic bounded model checking (Sec. 2.4), use function summaries for a modular description of components common to several tasks (Sec. 2.5.1), adapt SMT reasoning framework for a high-level description of a model of the program (Sec. 2.3), and often choose to abstract any non-elementary operators and functions description from the model (Sec. 2.4.2). We then describe different techniques to refine the first-order formula we initially construct based on the general steps above.

# Chapter 3

# Background

In this chapter, we discuss related work in the area of software verification and model checking relevant to the work in this thesis in general. We start with a review of the evolution of efficient model checking approaches (BDDs, symbolic model checking, SAT solvers, and SMT solvers). Then we discuss in more details the main techniques used in this thesis, as incremental approaches in verification, function summaries for verification and abstraction refinement techniques in verification. We include a related work section to discuss the background of each of the refinement techniques in Chap. 6, Chap. 7, and Chap. 8.

Model checking [EC80, CE82, QS82, CES86, CGP99] was developed independently in the 1980s by Clarke and Emerson [EC80, CE82], and by Quielle and Sifakis [QS82]. Different classes of methods of model checking have been developed since then to better deal with the *state explosion problem* [CGJ$^+$01, CKNZ12] and with the *computability problem* [CGP99] in software model checking [QS82, LP85, CES86, Cla97, JM09]. These have included various methods and techniques for checking, modelling and specification.

In model checking, we can describe the specification (or system requirements)

with a logical formula. Initially, the temporal model-checking algorithm [CE82, QS82] used *temporal logic specification* [MO81, Boc82, KdR85, CES86, FCSM93] for describing the requirements of the system. With the appearance of model checking algorithms without BDDs [BCCZ99, BCC+99, BCC+03] originally with SAT solvers and later on, with SMT solvers [DNS05]. The input for model checking algorithms has described the specification in temporal logic, propositional logic and first-order logic. In this thesis, the system requirements are safety properties and are encoded as a first-order formula.

The original model checking algorithm was an explicit state algorithm [CES86, CGP99, CG18], usually for hardware verification, though the method has also been applied for software verification [HL91, Hol97, Ios01, ELLL04, JM04, BBČ+06b, BEG+07, Hol11, vRAR11, BBH+13, CPR13].

*Symbolic model checking* was introduced later to handle the state explosion problem better and has been able to perform model checking tasks with a large number of states (initially, $10^{20}$ states and beyond) [BCM+92, McM93b]. The algorithm [BCM+92, McM93a] was originally presented with *Binary Decision Diagrams* (BDDs) [Bry86]; e.g., the SMV model checker [McM93a], IBM's model checker RuleBase [BBEL96, BBDE+97], Bebop [BR00], Moped [ES01], Uppaal2k [DA02] (with BDD's extension for real-time systems), and Rabbit model checker [BLN03]. Symbolic model checking without BDDs proposed using SAT solvers instead of BDDs and presented the idea of SAT-based bounded model checking in the late 90s [BCCZ99, BCC+99, CBRZ01, BCC+03] and SMT-based bounded model checking almost a decade later [AMP06, AMP09]. An interesting research direction of SAT-based symbolic model checking was investigating the benefits of integration with BDDs [CCG+02, CGP+02] and presenting unbounded symbolic model checking [McM02].

In this thesis, we have used model checking techniques and created a model checker described in Chap. 4 with all the additional refinement approaches we introduced in Chap. 5-8 in the context of SMT-based symbolic bounded model checking. We focus in the next paragraphs only on the related work in symbolic bounded model checking and additional techniques for efficient software verification. To better understand the scope of research related to software verification, we discuss the verification tasks in software before that, briefly.

Model checking has been applied successfully to create reliable systems for different verification problems in hardware [BCMD90, CLM91, CGH+93, JR11, Bon16], embedded software [CC97, God97, SS01, DA02, BLN03, CFMS12] as real-time systems [CC97, DA02, BLN03], satellite onboard software [AV14, GDH14], robotic systems [God97, LVB+12, FBZ+18] and aviation [Cla97, GH02, TB08, BVWW09, CGH+15], and general algorithmic software problems.

General algorithmic software [WSH+08, Com19, lib19, The19b] has been linked to a wide range of software verification tasks such as software verification of concurrency and parallelism [QS82, LP85, CES86, GW93, Pel96, BGP99, RDH03, AQR+04, QR05, RG05, BAM06, BBC+06a, WBKW07, VYY09, MMN+12, AKT13, WKO13, Hua15, GKQT18], data structures and data-types (e.g., arrays, bit-vectors, strings, integers and floats, and heaps) [BGP99, Ios01, RDH03, IYG+05, PW05, JM07, KST+08, BB09, KST+09, VYY09, ABG+12, CFMS12, KT14, CGI+18a], loops, recursion, and control flow in procedural programs [AY01, BR02b, ABE+05, AAB+07, KST+08, KST+09, DHKR11, SFS12a, SFS12b, FS15, KGC16, ABE18], and buffer over-flow and under-flow detection, NULL pointer dereferencing (and pointer-arithmetic in general) and memory safety problems [BHJM05, BHMV05, CGM05, IYG+05, QR05, SISG06, JM07, KST+08, KST+09, VYY09, MMN+12, WKO13, KT14, Hua15].

In this thesis, we focused on verification problems in software and embedded software. This included verification of code of drivers and operating system utilities and general algorithmic software. The verification problems were related to different data types (bit-vectors, integers and floats), loops and recursion (in the context of bounded model checking and summaries), and control flow problems. In Chap. 4, we also dealt with over-flow and under-flow detection. The benchmarks were written in C and were either crafted or taken from online sources, as SV-COMP [Com16, Com18].

In the rest of this chapter, we focus on describing the recent research in software model checking (focusing on bounded and symbolic model checking only) and additional techniques to deal with the state explosion problem. We limit the discussion here to techniques used in the thesis, that is, incremental verification and abstraction (SMT modelling and function summaries).

**Bounded model checking.** Bounded model checking (BMC) was presented originally in the late 90s by Biere et al. [BCCZ99, BCC$^+$99] as a complementary technique to symbolic model checking with BDDs. It relied on SAT solvers to exhaustively check the system up to a given limited depth. Hence, BMC is inherently incomplete [VWM15, BK18]. With additional techniques, the technique is complete [VWM15, BK18], as $k$-induction [SSS00, DHKR11, BDW15, RICB17, WDH19], interpolation [McM03, CMNQ06, CPP14], abstraction refinement techniques [PBG05, CPP14, BDW15], and inductive techniques with iterative strengthening [Bra11, EMB11]. In our work, we have adopted BMC as an approximation of the verification problem with no additional technique.

The bounded model checking technique proposed using SAT solvers [BCCZ99, BCC$^+$99, CBRZ01, BCC$^+$03] and later using SMT solvers [AMP06, AMP09], in-

stead of BDDs to overcome the memory bottleneck required for storing and manipulating BDDs. However, verification techniques with SAT solvers, in general, were already in use earlier (e.g., [SS90]); the idea of solving path constraints with SAT solvers has been used before for planning [KS96]. The SMV model checker [BCCZ99, BCC+99] was the first model checker to include an implementation of BMC for LTL formulas which later extended to CTL, LTL and PLTL formulas in NuSMV [CCG+02].

Bounded model checking techniques have been using propositional logic for modelling and specification description (e.g., [CKL04, CKSY05, KT14, SFS12a, SFS12b, FCHS15]) and first-order logic with the advent of SMT solvers for software verification [DNS05]. In our work, we integrated the old model checker FunFrog for propositional logic support while adding a new set of algorithms for modelling and specification with first-order logic; we implemented SMT-based BMC, a technique which has been already used before, for example in [AMP06, AMP09, CFMS12, RE14, CdLF16]).

In software BMC, we use the *control flow graph* (CFG) representation to compute the transition relation with a transition system defined over the whole program. We convert each block of statements into SSA and represent all the memory operations as defined in hardware [DKW08].

We used in HiFrog the implementation of CPROVER framework for C code for the flow described above, similarly to many other tools as CBMC, SATabs, ESBMC, and JBMC. CBMC was the first implementation of BMC for C programs [KT14]. TCBMC, a SAT-based BMC, is a CBMC version for concurrent programs developed by IBM [RG05], while Esbmc is an SMT-based context-BMC with $k$-induction and invariants algorithms [CFMS12]. FunFrog, a SAT-based BMC with function summaries for incremental verification, has applied a similar

approach to our work [SFS12a, SFS12b]. Last, JBMC is a new bounded model checking tool for verifying Java byte-code [CKK+18]. Our tool HiFrog is an SMT-based BMC for sequential C programs with several additional techniques: incremental solving, incremental verification with SMT summaries, and CEGAR. While the majority of the tools has had support for CEGAR and other techniques for refinement as well as incremental solving, none of the methods has applied all the three techniques in the context of SMT-based BMC.

**Satisfiability Modulo Theories (SMT).** The *satisfiability modulo theories* (SMT) [DNS05] reasoning framework checks the satisfiability of first-order formulas (usually in the SMT-LIB2 format [BST10]). The framework models the problem in Boolean formulas a while expressing domain-specific knowledge with first-order theories (e.g., arrays, fixed-size bit-vectors, uninterpreted function, arithmetic and quantifiers). SMT solvers were usually built on top of a SAT solver as Grasp [SS96], Chaff [MMZ+01], zChaff [ZM02, Zha03], MiniSAT2 [ES04] and SATO [Zha97], or an SMT solver as MathSAT5 [CGSS13] and Z3 [DMB08b].

SAT solvers have applied the modern implementation of the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [DP60, DLL62] for deciding the satisfiability of a propositional logic formula. In this framework, the SMT solver repeatedly tried to find a satisfying valuation for the SAT formula and checked consistency under the domain-specific theory using a theory solver usually with lazy or eager approaches. Commonly, SMT solvers implemented the DPLL algorithm extension, that is, the DPLL(T) framework [GHN+04, NOT06, NO05] for DPLL-based SMT solving. For example, the SMT solvers Z3 [DMB08b] and MathSAT5 [CGSS13] used this framework in their implementation. Lazy approaches have maintained

a tight integration of the Boolean reasoning of a DPLL-style search with theory-specific solvers; in Chap. 7, we implemented an eager approach, as the algorithm flatten the problem directly to the main SAT solver.

Common SMT solvers for verification have included, for example, the following tools: OpenSMT2 [HMAS16], Z3 [DMB08b], CVC4 [BCD+11], MathSAT5 [CGSS13], raSAT [TVKO17], dreal [GKC13] and iSAT3 [FHT+07]. HiFrog implementation was mainly depending on the OpenSMT2 solver for solving and interpolating. We sometimes used the Z3 solver for theory combination and full incremental verification solving (push and pop).

**Incremental solving.** Incremental SAT and SMT solvers [Hoo93, WKS01, ES04, DMB08b, BPST10, BCD+11, CGSS13, NRS14, HMAS16] have implemented different approaches and allowed to alter the constraints in the solver via push and pop operations in between two different checks for satisfiability. These approaches have assisted in improving performance and dealing with more complex problems, mainly when required to solve a set of related formulas [Sht01, WKS01, ES03, CLM+10, EMA10, FCN+10, Wie14, SKB+17].

In this thesis, we have utilised the incremental solving support of the SMT solver only when adding new assert statements, except in LB-CEGAR. In Chap. 6, the implementation of LB-CEGAR has leveraged incremental solving techniques in the solvers to speed-up the solving time between two refinement cycles of (same or contained) sub-domains of values and avoid initialising the solver once starting the refinement of a new sub-domain of values.

**Incremental verification.** Incremental verification has been extensively researched in domains such as hardware verification, deductive verification, and model checking. The CPAchecker [CPA19, BK11] tool has been able to mi-

grate predicates across program versions [BLN+13]. Deductive verification tools such as VIPER and DAFNY offered modular verification [MSS16] and caching the intermediate verification results [LW15] respectively.

CBMC [KT14, SKB+17] is a symbolic bounded model checker for C that to a limited extent exploited incremental capabilities of a SAT solver but did not use or output any reusable information like function summaries. Similar to HIFROG, ESBMC [CFMS12] also shared the CPROVER [CKL04, cpr19] infrastructure and is based on an SMT solver. To the best of our knowledge, it did not support incremental verification [CdLF16]. FUNFROG [SFS12a, SFS12b], the model checking tool HIFROG built on top of, used an approach for extracting and reusing interpolation-based function summaries in the context of SAT-based bounded model checking. Unlike HIFROG, the work in FUNFROG focused only on propositional logic and did not consider the rich field of first-order theories available in modern SMT solvers. Hence, despite behaving incrementally, FUNFROG was expensive in many cases in practice. KIND [KGTW12, GBW+18] model checker has dealt with verification of multiple safety properties of synchronous systems by allowing the simultaneous verification of multiple properties incrementally. That is, when each property has proven valid, the tool immediately used its invariant to aid the verification process. Unlike KIND, HIFROG has not been supporting simultaneous verification of properties; however, we find this work interesting and relevant to the work in this thesis, especially since OPENSMT2 had recently applied algorithms for parallel and Distributed solving [HMAS16, MHS18].

**Function summaries and interpolation.** Function summaries date back to Hoare logic [Hoa71], where a pair of pre-condition and post-condition can be seen as an over-approximating function summary. Function summaries have been com-

puted using algorithms for constructing Craig interpolants [Cra57, Hua95, Kra97, Pud97, McM05, DKPW10, RAF+13], symbolic path formulas [God07, AGT08], data-flow analysis [RHS95, BR00, BKW07], and iterative discovery of modification of variable values, used in model checkers SATURN [XA05] and CALYSTO [BH08]. In this thesis, function summaries are either extracted manually (i.e., user-defined summaries) or computed via Craig interpolants.

Interpolation-based function summarization framework has already been applied in static and dynamic analysis and in software verification [HHP10, McM10, AGC12, SFS12a, SFS12b, SSCS12, KHK19]. FUNFROG [SFS12a, SFS12b] used over-approximated function summaries constructed via Craig interpolation in a SAT-based incremental checking framework, and EVOLCHECK [FSS13] applied a similar approach for SAT-based incremental upgrade checking. FOCAL [KHK19] constructed and refined under-approximate of function summaries using Craig interpolants as refining constraints for verification via concolic testing. In this thesis, we used over-approximated function summaries as first-order formulas to aid the verification process in incremental model checking framework.

McMillan introduced the first application of interpolants in formal verification [McM03]. Since then, interpolation has been applied in algorithms with various extensions in model checking [CMNQ06, EKS06, McM06, VG09, HHP10, KW11, ABG+12, AM13, RHK13, CPP14, McM14, VGM15, FSS17, FB18, IX18, IX19]. The model checkers CPACHECKER [BK11], SEAHORN [GKKN15], ULTIMATE AUTOMIZER [HCD+18] and others, leveraged interpolants in some form. In this thesis, we leveraged interpolants only for constructing function summaries from a successful verification of valid properties; interpolants for first-order formulas were the actual SMT summaries used in our framework.

In this thesis, we applied different interpolation algorithms as means of abstrac-

tion. These interpolation algorithms constructed interpolants of different *strength* and *size* for different theories via the LIS framework [DKPW10]. The **labelled interpolation system** (LIS) framework computes interpolants given a refutation and a labelling function [Hua95, Kra97, Pud97, McM05, DKPW10, RAF+13]. In this thesis, we used the LIS framework in [DKPW10], including the *Proof-Sensitive* interpolation algorithms [AFHS15] for propositional logic interpolation. We used a EUF interpolation algorithm with *duality-based interpolation* [AHAS17]; a similar extension was applied to the interpolation algorithm for LRA based on [McM05]. LIA had no interpolation algorithm in HiFrog.

**Abstraction and abstraction refinement.** Verification by abstraction is a class of techniques for verifying large systems and software (e.g., [CC77, CC79, CGL94, CIY95, LGS+95, GS97, CU98, CGJ+00, CCK+02, CGJ+03, CKSY05, GS05, BBB+10, EMA10, CCM12, WKO13, RNO14, HCR+16, KIY16, DG18]). Abstraction in model checking [CGL94, DG18] has intended to reduce the size of the model [DGG93, CGL94] for verifying larger systems (e.g., [CGL94, BBB+10, SKB+17]). These approaches traded precision for efficiency, which caused false results in case the abstraction failed to capture the relevant behaviour required either for finding the bug or proving correctness.

In model checking, this problem has been dealt with abstraction refinement (e.g., [CGJ+00, CCK+02, CGJ+03, CKSY05, GS05, EKS06, JM07, KKNP09, RNO14, LMN15, DG18]). The counterexample-guided abstraction refinement (CEGAR) [CGJ+00, CGJ+03] is a common and successful approach for abstraction-refinement in model checking [BR02a, CKSY05, GS05, EKS06, JM07, WBKW07, HWZ08, BLR11, KSU11, OR11, LMN15, HM19]. Iterative refinement of the abstract model to eliminate the current counterexample, was proposed as

localization reduction in [Kur94, AIKY95], and only later generalised into the CEGAR approach [CGJ+00, Saï00, BR02a, CGJ+03]. We applied the CEGAR approach in this thesis in all the refinement algorithms we have presented, sometimes with slight modifications.

SLAM [BR02a, BLR11] was the first model checker implemented CEGAR approach combined with BDD-based model checking [BR00, BR02a]. The BLAST model checker implemented lazy abstraction, which is an optimization of CEGAR [CCG+04, BHJM07]. The F-SOFT [IYG+05] model checker combined CEGAR-based predicate abstraction refinement with other verification techniques to check standard runtime errors in C programs as buffer overflows, and null dereferences. Other tools combined CEGAR with SAT-based model checking as MAGIC [EKS06] and SATABS [CKSY05]. We applied CEGAR in an SMT-based verification approach similar to [HWZ08, Arm09, BW12, CNR13, LMN15, BD16, CGI+17b, CGI+18b, WDH19]. More precisely, we implemented CEGAR in an SMT-based incremental BMC.

# Chapter 4

# SMT-based Function Summarization for Software Verification

Function summarization can be used as a means of incremental verification based on the structure of the program. In this chapter, we present HiFrog, a function-summarization-based model checker that uses SMT as the modelling and summarization language.

The HiFrog model checker supports four encoding precisions through SMT: uninterpreted functions, linear real arithmetic, linear integer arithmetic, and propositional logic. In addition, the tool allows an optimized traversal of reachability properties, counterexample-guided summary refinement, summary compression, and user-provided summaries. We describe the use of the tool through the description of its architecture and a rich set of features. The description is complemented by an experimental evaluation on the practical impact the different

SMT precisions have on model checking. The tool together with a comprehensive demo is available at `http://verify.inf.usi.ch/hifrog`.

## 4.1 Introduction

*Incremental verification* addresses the unique opportunities and challenges that arise when a verification task can be performed in an incremental way, as a sequence of smaller closely related tasks. We present an implementation of the incremental verification of software with assertions that uses the insights obtained from a successful verification of earlier assertions. As a fundamental building block in storing the insights we use function summaries known to provide speed-up through localizing and modularizing verification [SFS12a, RAF+13].

We describe in this chapter the HiFrog verification tool that uses Craig interpolation [Cra57] in the context of Bounded Model Checking (BMC) [BCCZ99] for constructing function summaries. The novelty of the tool is in the unique way it combines function summaries with the expressiveness of satisfiability modulo theories (SMT). The system currently supports verification based on the quantifier-free theories of LRA and EUF, in addition to propositional logic (BOOL).[1] Compared to our earlier propositional tool FunFrog [SFS12a], the SMT summaries are smaller and more efficient in verification. They are also often significantly more human-readable, enabling their easier reuse, as well as injection of summaries provided directly by the user. The difference is due to the propositional summaries being based on correctness proofs over circuit-level representation of arithmetic operations. Theory encoding instead directly uses arithmetic symbols in the summaries. In addition, the tool offers a rich set of features such as verification of

[1] There is a partial implementation of modelling LIA in HiFrog but without the functionality described in this chapter, hence omitted.

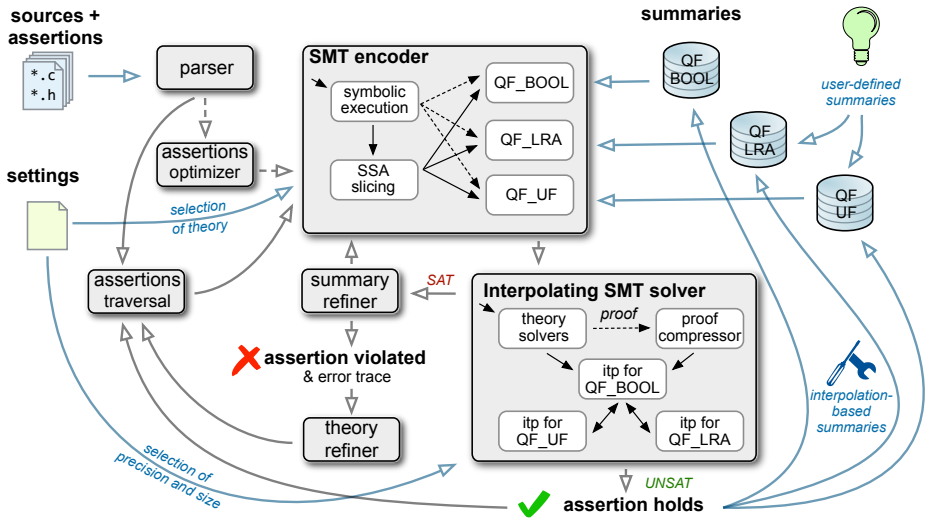


**Figure 4.1:** HiFrog: tool overview.

recursive programs, different ways of optimizing the summaries with respect to both size and strength, efficient heuristics for removing redundant safety properties, and easy-to-understand witnesses of property violations that can be directly mapped to bugs in the source code.

## 4.2 Tool Overview

HiFrog consists of two main components *SMT encoder* and *interpolating SMT solver*, and the function *summaries* described in Fig. 4.1, where the grey and black arrows connect different modules of the tool (dashed - optional) and the blue arrows represent the flow of the input/output data. The components are initially configured with the theory and the interpolation algorithms. The tool then processes assertions sequentially using function summaries when possible. The results of a successful assertion verification are stored as interpolated function summaries, and failed verifications trigger a refinement phase or the printing of an error trace. This section details the tool features.

### 4.2.1 Preprocessing

The source code is parsed and transformed into an intermediate *goto-program* using the GOTO-CC [cpr19] symbolic compiler. The loops are unwound to the pre-determined number of iterations. HIFROG identifies the set of assertions from the source code, reads the user-defined function summaries (if any) in the SMT-LIB2-format, and makes them available for the subsequent analysis.

### 4.2.2 SMT encoding and function summarisation

For a given assertion, the goto-program is *symbolically executed* function-per-function resulting in the "modular" Static Single Assignment (SSA) form [CFR+89, CFR+91] of the unwound program, i.e., a form where each function has its own isolated SSA-representation. To reduce the size of the expression in an SSA form, HIFROG performs backward *SSA-slicing* that keeps only the variables in the SSA form that are syntactically dependent on the variables in the assertion.

When the SSA form is pruned, HIFROG creates the SMT formula in the pre-determined logic (BOOL, EUF, LRA or LIA). The modularity of the SSA form comes in handy when the function summaries of the chosen logic (either user-defined, interpolation-based, or treated non-deterministically) are available. If this is the case, the call to a function with the available summary is replaced by the summary. The final SMT formula is pushed to an SMT solver to decide its satisfiability.

Due to the over-approximating nature of function summaries, the program encoded with the summaries may contain spurious errors. The *summary refiner* identifies and marks summaries directly involved in the detected error; HIFROG

returns to the encoding stage to replace the marked summaries by the precise (up to the pre-determined logic) function representations. Note that due to refinement, HiFrog reveals nested function calls (including recursive ones) which are again replaced by available summaries. For an unsatisfiable SMT formula, HiFrog extracts function summaries using interpolation. The extracted summaries are serialized in a persistent storage so that they are available for other HiFrog runs.

For a more detailed description regarding function summaries in bounded model checking, see [SFS12a, SFS12b]. Currently HiFrog supports three encoding precisions for interpolant-based function summaries: Uninterpreted Functions (QF_UF), Linear Real Arithmetic (QF_LRA), and Propositional Logic; Linear Integer Arithmetic (QF_LIA) has a partial support in HiFrog. Examples of function summaries with detailed technical explanations are available at [HiF17b].

### 4.2.3 User-defined summaries in HiFrog

We allow incorporating function summaries into the verification process of HiFrog. Above we described function summaries that are *Craig interpolants* [Cra57] from one of the previous iterations of model checking. We also allow the user to supply *user-defined* summaries, based on their external knowledge of the system.

User-defined summaries follows the same form of function summaries, we treat function summaries and user-defined summaries in the same way when loading the summaries and while using these summaries for the current verification task. It is possible to provide to HiFrog a *library of user-defined summaries*; the whole set of summaries is uploaded to the SMT solver at once where each of the summaries is treated in the same way as any function summaries by the SMT solver.

We model user-defined summaries with the quantifier-free SMT theories for equality logic with uninterpreted functions (EUF), linear integer arithmetic (LIA), and linear real arithmetic (LRA). We refer both, function summaries and user-defined summaries as SMT summaries when modelled with SMT logics.

### 4.2.4 Theories

HiFrog supports four different quantifier-free theories in which the program can be modelled: bit-precise BOOL, EUF, LIA, and LRA. The use of theories beyond BOOL allows the system to scale to larger problems since encoding in particular the arithmetic operations using bit-precision can be very expensive. As the precise arithmetics often do not play a role in the correctness of the program, substituting them with linear arithmetics, uninterpreted functions, or even non-deterministic behaviour might result in a significant reduction in model checking time (see Sec. 4.3). If a property is proved using one of the light-weight theories EUF, LRA and LIA, the proof holds also for the exact BMC encoding of the program. However, the loss of precision can sometimes produce spurious counterexamples due to the over-approximating encoding. The light-weight theories therefore need to be refined (that is, using *theory refiner*) to BOOL if the provided counterexample does not correspond to a concrete counterexample.

In Chap, 7, we present a process to refine the precision automatically.

### 4.2.5 Obtaining summaries by interpolation

HiFrog relies on different interpolation frameworks for the different theories it supports. As a result, the generation of propositional, EUF and LRA interpolants can be controlled with respect to strength and size by specifying an interpolation algorithm for a theory; LIA has currently no interpolation algorithm available

in HiFrog. For propositional logic we provide the *Labelled Interpolation Systems* [DKPW10] including the *Proof-Sensitive* interpolation algorithms [AFHS15]. Interpolation for EUF is implemented with *duality-based interpolation* [AHAS17], and a similar extension is applied to the interpolation algorithm for LRA based on [McM05]. HiFrog also provides a range of techniques to reduce the size of the generated interpolants through removing redundancies in propositional proofs [RAF+13]: the algorithms *RecyclePivotsWithIntersection* and *LowerUnits*, structural hashing, and a set of local rewriting rules.

### 4.2.6 Assertion optimizer

In addition to incremental verification of a set of assertions, HiFrog supports the basic functionality of classical model checkers to verify all assertions at once. For the cases when the set of assertions is too large, it can be optimized by constructing an *assertion implication relation* and exploiting it to remove redundant assertions [FCHS15]. In a nutshell, the assertion optimizer considers pairs of spatially close assertions $a_i$ and $a_j$ and uses the SMT solver to check if $a_i$ conjoined with the code between $a_i$ and $a_j$ implies $a_j$ (if there is any other assertion between $a_i$ and $a_j$ then it is treated as assumption). If the check succeeds, then $a_j$ is proven redundant and its verification can be safely skipped.

## 4.3 HiFrog Usage

We provide a Linux binary of HiFrog reading as input a C-program, assertions to be verified, a set of parameters and the interpolated or user-defined function summaries in the SMT-LIB2 format.

HiFrog exploits the CPROVER framework [CKL04] and inherits some of

its options (e.g., `-unwind` for the loop unrolling, `-show-claims` and `-claim` for managing the assertions checks); the ability for the user to declare and to use a `nondet_TYPE()` function of a specific numerical type (e.g., int, char, long, double, unsigned, in LRA and int, char, long, in LIA) or add a `__CPROVER_assume()` statement to limit the domain to a specific range of values.

HiFrog uses EUF by default but can be switched to LRA, LIA or propositional logic via the `-logic` option. HiFrog uses a variety of interpolation and proof compression algorithms to control the precision (with `-itp-uf-algorithm` option for EUF, `-itp-lra-algorithm` option for LRA, and `-itp-algorithm` option for propositional interpolation) and the size (with `-reduce-proof`) of summaries. The summary storage is controlled using the `-save-summaries` and `-load-summaries` options. In between verification runs, the summaries contained in the corresponding files for EUF and LRA might be edited manually. Finally, HiFrog supports the identification and reporting of redundant assertions with `-claims-opt`, a useful feature for some automatically generated assertions [FCHS15].

At the end of each verification run, HiFrog either reports `VERIFICATION SUCCESSFUL` or `VERIFICATION FAILED` accompanied by an error trace. An error trace presents a sequence of steps with a direct reference to the code and the values of variables in these steps. In most cases when EUF and LRA introduce a spurious error, HiFrog outputs a warning, and thus the user is advised to use HiFrog with a more precise theory. HiFrog also reports the statistics on the running time and the number of the summary-refinements performed.

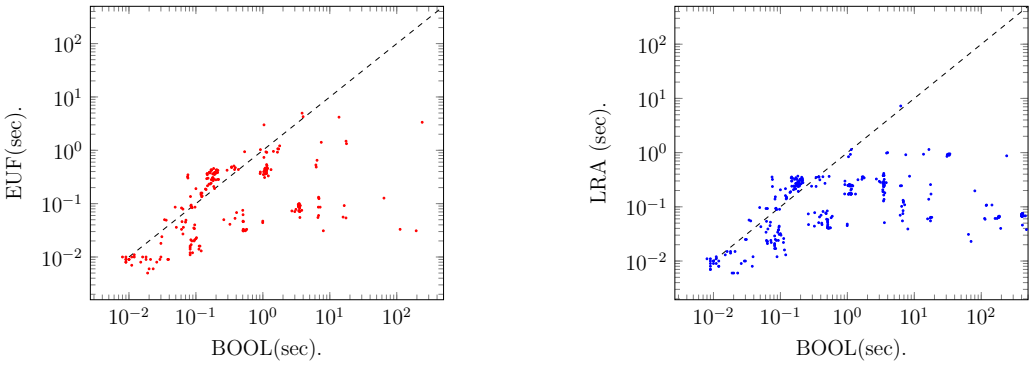


**Figure 4.2:** Running time by BOOL against EUF and LRA.

## 4.4 Experimental Results

We evaluated HiFrog on a large set of C programs coming from both academic and industrial sources such as SV-COMP [Com16]. All benchmarks contained multiple assertions to be verified. To demonstrate the advantages of the SMT-based summarisation, we provide data for analysis of benchmarks containing 1086 assertions from which 478 were proven to hold using BOOL (meaning that those properties satisfy the system specifications with propositional logic and it is the highest number of instances we can prove to hold using the current implementation of HiFrog with BOOL defined to be the most precise theory).

Fig. 4.2 presents two logarithmic plots for comparison of running times of HiFrog with BOOL to respectively EUF and LRA. Each point represents a pair of verification runs of a holding assertion with the two corresponding theories using the interpolation-based summaries. For most of the assertions, the verification with EUF and LRA is an order of magnitude faster than the verification with BOOL. Even despite the over-approximating nature of EUF and LRA, our experiments witnessed a large amount of properties which were also proven to be

**Table 4.1:** Benchmarks UNSAT-solved assertions with BOOL, LRA and UF.

| Benchmark | Asserts Solved with BOOL Only | Assert Solved with BOOL and UF | Assert Solved with BOOL and LRA |
|---|---|---|---|
| token.c | 34 | 34 | 34 |
| s3.c | 27 | 19 | 22 |
| mem.c | 97 | 97 | 97 |
| disk.c | 23 | 7 | 15 |
| ddv.c | 143 | 47 | 47 |
| cafe.c | 30 | 15 | 20 |
| tcas-asert.c | 30 | 17 | 30 |
| P2P.c | 94 | 8 | 67 |
| Total (8 cases) | 478 (100%) | 244 (51.05%) | 332 (69.45%) |

correct by employing the light-weight theories of HiFrog (namely, 51.05% and 69.45% of validated properties out of 478 for EUF and LRA respectively) [2].

We applied statistical tests to more rigorously analyse the results above.

**Statistical Tests of the Efficiency of HiFrog.** We hypothesised that assertions proved to hold by BOOL and HiFrog (with EUF or LRA) had no significant difference in terms of time consumption (our null hypothesis). We focused on the UNSAT results when analysing our tool efficiency because it would be a challenging target to prove safety than to find a counterexample with first-order logic in comparison to propositional logic. We justified our choice of the null hypothesis when we considered both the quality of HiFrog's implementation and the characteristic of SMT-solvers in general; on the one hand, we evaluated HiFrog in this chapter with its early implementation known for its poor performance (time and memory), but on the other hand, SMT-solvers are known to speed up computations in comparison to SAT solvers.

We applied the statistical test on eight benchmarks with multiple assertions

[2]From the evaluation of light-weight theories in this chapter, we omitted the reference to LIA. Since, it currently has no interpolation nor function summaries support in OpenSMT2 and HiFrog.

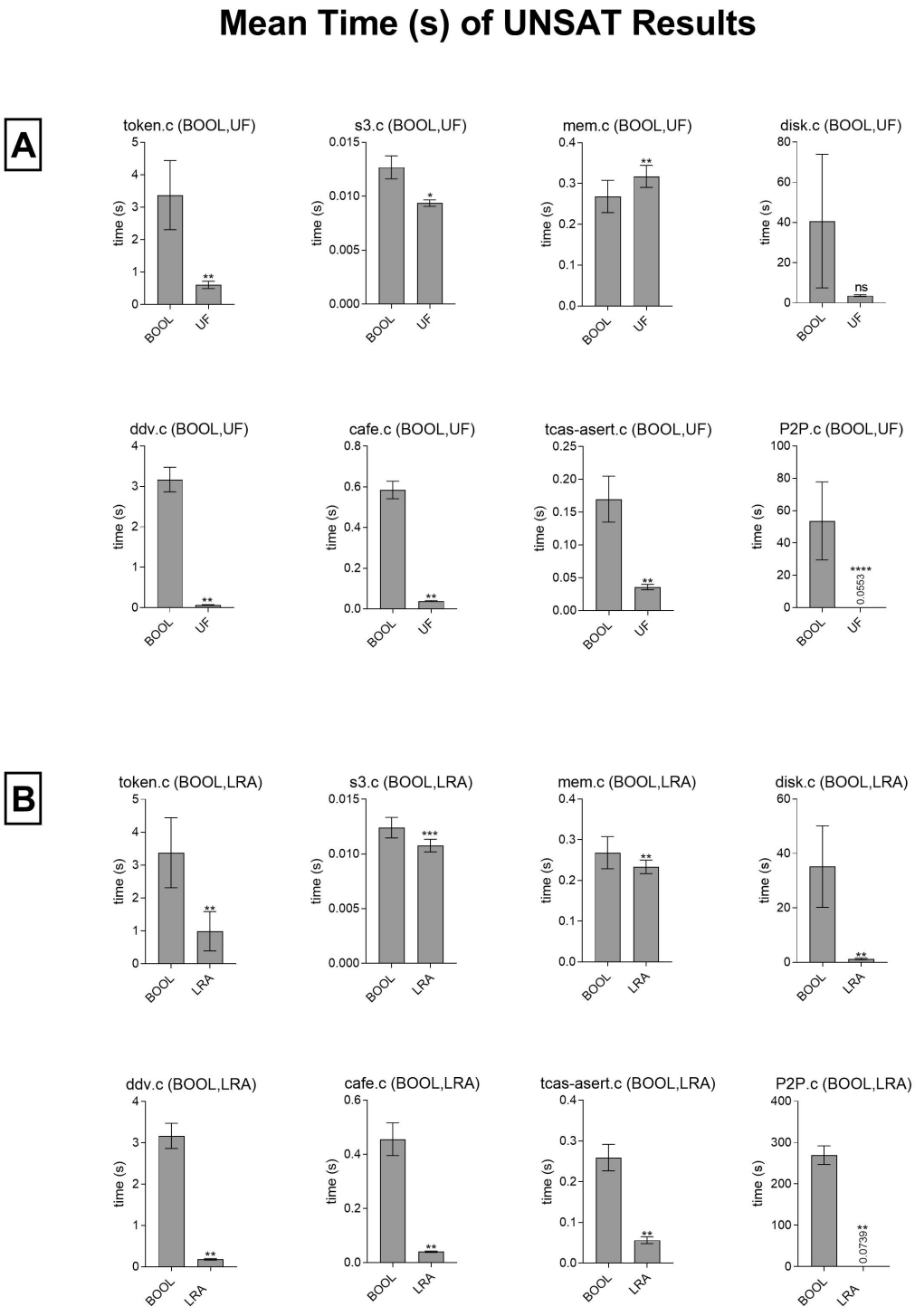


**Figure 4.3:** Graphical description of statistical analyses with Wilcoxon test on reducing the solving time of pair of assertions in benchmarks, presented as histograms of means ± SEM and labels: (**)P $< 0.0001$, (*)P $= 0.0018$, (***)P $= 0.0088$, (****)P $= 0.0078$ and (ns)P not significant. **A**: (EUF,BOOL) and **B**: (LRA,BOOL).

(the exact number of assertions are in Table 4.1) for performance comparison of UNSAT results, and we sampled the runtime in second. The majority of the data sets did not pass the normality test; all failed except three sets of samples: disk.c (EUF), cafe.c (LRA) and P2P.c (EUF). Therefore, we applied the Wilcoxon test (a non-parametric t-test). Our goal was to test which method was faster, that is, the method with the smaller mean value, when there was a significant statistical difference (per benchmark).

Figure 4.3 presents the results of the statistical analyses with Wilcoxon test (fully-paired, two-tailed) as histograms of the mean value (in seconds) with Standard Error Mean (SEM), where statistically significant p-value at $\leq 0.05$, label **A** for EUF and BOOL's comparison, and label **B** for LRA and BOOL's comparison. The results of HiFrog with either EUF or LRA are always on the right bar.

We observed a signification difference in all sets in **A** and **B** except `disk.c` with (EUF, BOOL), where for all benchmarks the mean value of solving time of an assertion was smaller with HiFrog (EUF, LRA) than with BOOL except for `mem.c` (EUF, BOOL). SEM was relatively small in all histograms' right bars (EUF and LRA) except `token.c` with (LRA, BOOL), unlike the left bars (`token.c`, `disk.c`, `tcas-asert.c` with (EUF, BOOL), and `P2P.c` with (EUF, BOOL)), which might indicate some sort of a common-case optimization implementation in BOOL. To summarize the results, six out of eight benchmarks in set **A** and all eight benchmarks in set **B** demonstrated a significant reduction in the average-time of proving assertion correctness.

The analysis of those experiments' results revealed that model checking using the EUF and LRA-based summarisation was extremely efficient in proving correctness of assertions.

**Experiment settings.** The timing results were obtained on an Ubuntu 14.04.1 LTS server running two Intel($R$) Xeon($R$) $E5620$ $CPUs@2.40GHz$ and 16GB RAM. We prepared a pre-compiled Linux-binary available at the Virtual Machine at `http://verify.inf.usi.ch/hifrog/binary`; our benchmarks set is available at `http://verify.inf.usi.ch/hifrog/bench` and can facilitate the property verification for other researchers.

# Chapter 5

# Lattice-based Counterexample-Guided Abstraction Refinement in Bounded Model Checking

In this, chapter we present an algorithm for bounded model checking with SMT solvers of programs with library functions — either standard or user-defined. Typically, if the program correctness depends on the output of a library function, the model checking process either treats this function as an uninterpreted function or is required to use a theory under which the function in question is fully defined. The former approach leads to numerous spurious counterexamples, whereas the latter faces the danger of the state-explosion problem, where the resulting formula is too large to be solved by means of modern SMT solvers.

We extend the approach of user-defined summaries and propose to represent the set of existing summaries for a given library function as a *lattice* of subsets of

summaries, with the meet and join operations defined as intersection and union, respectively. The refinement process is then triggered by the lattice traversal, where in each element the SMT solver uses the subset of SMT summaries stored in this element to search for a satisfying assignment. The direction of the traversal is determined by the results of the concretization of an abstract counterexample obtained at the current element. Our experimental results demonstrate that this approach allows solving a number of instances that were previously unsolvable by the existing bounded model checkers.

## 5.1 Introduction

SMT-based BMC amounts to verifying correctness of a given program within the given a bound on the maximal number of loop iterations and recursion depth, with the model and specification expressed in first-order logic. Successful verification of software relies on finding an expressive model to capture software's behaviours relevant to correctness but sufficiently high-level to prevent the reasoning from becoming prohibitively expensive — the process known as *theory refinement* [HAE+17]. However, balancing between performance and the model's level of abstraction is a challenging task, as more precise theories are usually more expensive computationally. Moreover, often there is no need to refine the theory for the whole program. As the modern approach to software development encourages modular development and re-use of components, programs increasingly use library functions, defined elsewhere. If the correctness of the program depends on the implementation of the library (or user-defined) functions, there is a need for a modular approach that allows us to refine only the relevant functions. Yet, currently, the theory refinement is not performed on the granularity level of a sin-

gle function, hence BMC of even simple programs can result in a state explosion, especially if the library function is called inside a loop.

In this chapter, we introduce an approach to efficient SMT-based bounded model checking with lattices of summaries for library functions. Each lattice contains summaries of a library function, and a summary is an equation or inequality of a library function property. Roughly speaking, the lattice is a *subset lattice*, where each element represents a subset of equations and inequalities that hold for some subset of inputs to the function; the *join* and *meet* operators are defined as union and intersection, respectively. The counterexample-guided abstraction refinement (CEGAR) [CGJ+00, CGJ+03] approach that we describe in this chapter is lattice-based, is triggered by a traversal of the lattice, and the CEGAR loop is repeated until one of the following outcomes occurs: (i) we prove correctness of the bounded program (that is, absence of concrete counterexamples), (ii) we find a concrete counterexample, or (iii) the current theory together with the summaries in the lattice is determined insufficient for reaching a conclusion.

The following motivational example illustrates the use of lattices with LIA (quantifier-free linear integer arithmetic) encoding and integer arithmetic theory.

**Example 7.** The code example in Fig. 5.1 describes the *greatest common divisor* (GCD) algorithm. We assume that both inputs are positive integers. The program is safe with respect to the assertion $g \leq x$. However, with the problem encoded with the first-order logic modulo Linear Integer Arithmetic (LIA), an SMT solver cannot prove the correctness of the program, as GCD is not expressible in linear arithmetic. The standard approach is to have $gcd(x, y)$ assume any integer value; thus, attempting to verify this program with an SMT solver with LIA results in an infinite number of spurious counterexamples.

```
1 int gcd(int x, int y)
2 {
3   int tmp;
4   while(y != 0) {
5     tmp = x%y;
6     x=y;
7     y=tmp; }
8   return x;
9 }
```

```
1 int main(void)
2 {
3   int x=45;
4   int y=18;
5   int g = gcd(x,y);
6
7   assert(g <= x);
8 }
```

**Figure 5.1:** The GCD program using *modulo* function.

In the example, we augment the solver with a set of *equations and inequalities* of properties of the *modulo* function, arranged in a lattice of summaries[1]. These equations and inequalities are taken from an existing set of lemmas and theorems of the Coq proof assistant [The19a] for $a\%n$:

$$f_1 \equiv z\_mod\_mult \equiv$$

$$\equiv a \ mod \ n = 0 \text{ with the assumption } a == x * n \text{ for some positive integer } x;$$

$$f_2 \equiv z\_mod\_pos\_bound \wedge z\_mod\_unique \equiv$$

$$\equiv (0 \leq a \ mod \ n < n) \wedge (0 \leq r < n \implies a = n * q + r \implies r = a \ mod \ n)$$

$$\text{for some positive integers } r \ \texttt{and} \ q, \text{ with the assumption } (n > 0) \wedge (a \neq x * n);$$

$$f_3 \equiv z\_mod\_remainder \wedge z\_mod\_unique\_full \equiv$$

$$\equiv (n \neq 0 \implies (0 \leq a \ mod \ n < n \vee n < a \ mod \ n \leq 0)) \wedge ((0 \leq r < n \vee n < r \leq 0)$$

$$\implies a = b * q + r \implies r = a \ mod \ n) \text{ with the assumption } \textbf{true}.$$

The assumptions are different from the original guards in [The19a], as these are

[1]The order of introducing properties is significant but different for each program. Since there is no set of properties, which all programs with modulo likely require for proving correctness, using one specific order for prioritizing some of the properties, is meaningless. Moreover, it can have a negative effect on the performance of the model checker if ignoring the size and complexity of the properties' expressions in the solver. On the other hand, our solution is general to all programs with modulo operator occurrences.

re-written during the construction of the lattice. Assumptions of elements can be re-written to eliminate the case where two or more assumptions of elements of a unique meet, refer to the same input value.

The original subset lattice consists of all subsets of the set $\{f_1, f_2, f_3\}$. It is analysed and reduced as described in Sec. 5.4 to remove equations and inequalities with contradicting assumptions[2] and equivalent elements. In this example, the set $\{f_3\}$ generalises $\{f_1\} \sqcup \{f_2\}$. Figure 5.2 shows the original subset lattice on the left, and the resulting meet semi-lattice on the right. Note that, we construct the meet semi-lattice of the modulo library function once and use the semi-lattice each time we verify a program with this library function. In the evaluation, we demonstrate the above with a larger set of summaries.

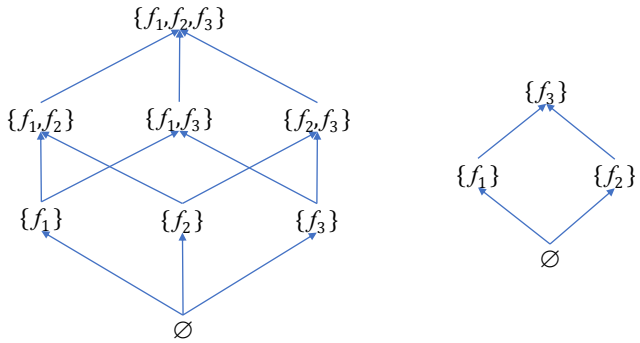


**Figure 5.2:** Diagrams of original subset lattice and reduced semi-lattice (*modulo* function).

In the lattice traversal, we start from the bottom element $\emptyset$ and traverse the semi-lattice until we either prove that the program is safe or find a real counterexample (or show that a further theory refinement is needed). In this example, we traverse the lattice until the element $\{f_3\}$, which is sufficient to prove that the

[2]For example: $assume(t > 0)$ and $assume(t < 0)$ are contradicting assumptions, since the conjunction of $(t > 0)$ and $(t < 0)$ is false.

program is safe. Specifically, the equation $f_1$ is used to prove loop termination, and the equation $f_2$ is used to prove the assert statement.

Our algorithms are implemented in the SMT-based bounded model checker HiFrog (Chap. 4, [AAC+17]) supporting a subset of the C language and using the OpenSMT2 SMT solver [HMAS16] and the Z3 [DMB08b] SMT solver. We demonstrate the lattice construction on sets of equations and inequalities of the *modulo* function. The summaries for the lattice construction are obtained from the built-in theorems and statements in the Coq proof assistant [The19a].

Our preliminary experimental results show that our approach can avoid the state-explosion problem and successfully solve programs that are not solvable using the standard CEGAR approach. Our set of benchmarks is a mix of our own crafted benchmarks and benchmarks from the software verification competition SV-COMP [Com18]. The lattices are constructed using data from an independent source, and we show that even with a relatively small lattice we can verify benchmarks which either are impossible to verify in less precise theories or are too expensive to verify with the precise definition.

The scripts, the source code, HiFrog tool, and lattices and programs used in our experiments, are available at [Git19a, Git19b, HiF19].

## 5.2 Preliminaries

We exploit the functionality of SMT function summaries by providing the SMT-based model checker HiFrog (Sec. 2.4.2 and Sec. 4) with a library of user-defined summaries (Sec. 2.5.1 and Sec. 4.2.3) to organize a set of equations and inequalities of properties of a library function in a lattice of guarded literals (literals are defined

in Sec. 2.3.1). We discuss the solution and its data structure in detail in the next section. See Sec. 2.1 for general definitions of a poset, a lattice and a semi-lattice.

## 5.3 Overview of the Solution

In this section, we describe our suggested lattice-based counterexample-guided abstraction refinement in bounded model checking (LB-CEGAR-BMC) approach for verification of programs with library functions. In Sec. 5.3.2, we discuss the general solution and focus on the modifications in the refinement loop in the original CEGAR approach (see Sec. 2.6.1, Fig. 2.14).

We discuss the lattice of guarded literals used in the LB-CEGAR-BMC approach in the next section.

### 5.3.1 Lattices of guarded literals

In this section we describe the construction of lattices of expressions for external functions.

We consider a subset lattice $SL(X)$ and $\langle SL(X), \sqcap \rangle$ and its reduce meet semi-lattice $\langle L, \sqcap \rangle$, for $X$ being a finite set of guarded expressions, as defined next.

A *guarded literal* is a Boolean expression describing some property of the function in question, together with the guard that defines a *continuous subdomain* of the inputs for which this property holds. For example, the property expressing the fact that for $0 < x < 2$, the value of $\sin x$ is positive is described by the guarded literal

$$(assume(0 < x < 2)) \wedge (\sin x > 0),$$

where $(0 < x < 2)$ is a *guard* of the *literal* $(\sin x > 0)$. Literals that hold for all $x$ (such as, for example, $(\sin x \leq 1)$ are guarded with $assume(\mathbf{true})$). A guard

cannot refer to a non-continuous domain. For example,

$$(assume(0 < x < 2) \vee (7 < x < 8)) \wedge (\sin x > 0)$$

is not a legal guarded literal in our framework.

Given a set of guarded literals $F$ for a library function $f$, the subset lattice $SL(F)$ consists of all subsets of these literals. However, it is easy to see that some elements in $SL(F)$ contain literals with contradictory guards. For example, a lattice of all subsets of $\sin x$ could contain the element $(assume(0 < x < 2)) \wedge (\sin x > 0)$ and the element $(assume(x = 0)) \wedge (\sin x = 0)$, which do not intersect on any subdomain of $x$. To reduce the size of the lattice and avoid unnecessary calls to the SMT solver, we reduce $SL(F)$ to a *meet semi-lattice* $L = \langle SL(F), \sqcap \rangle$ by removing all elements that have contradictory guards (that is, the conjunction of their guards is false). Note that, even if we remove an element, its properties appear in other elements that have no contradictory guards since $L$ is a subset lattice.

Note that after the removal of contradictory elements, the resulting set of subsets is no longer closed under union, but it is still closed under intersection, hence the resulting set is a meet semi-lattice. Note also that the resulting meet semi-lattice can have a set of maximal elements instead of the single maximal element. For brevity, in the rest of the chapter and the next chapter, we refer to the meet semi-lattice of guarded literals for a function $f$ simply as a lattice and use the notation $L$.

A *frontier* of a lattice $L$ is a set of elements $X(L)$ such that each chain from $\bot$ to a maximal element in $L$ intersects $X(L)$ in at least one element. The LB-CEGAR-BMC algorithm described in Sec. 5.5 and later also its generalisation,

the LB-CEGAR algorithm (Sec. 6.3), rely on the fact that the union of guards of each frontier of the lattice is the whole domain of the inputs. If this is not the case, we add elements to the lattice to cover the missing subdomains. For the example of $\sin x$ above, if we have only two guarded literals

$$(assume(0 < x < 2)) \wedge (\sin x > 0)$$

and

$$(assume(x = 0)) \wedge (\sin x = 0)$$

in our set, we add the guarded literals

$$((assume(x < 0)) \wedge \mathbf{true})$$

and

$$((assume(x \geq 2)) \wedge \mathbf{true})$$

to the set to cover the whole domain of $x$ (recall that the guards should refer to continuous subdomains, hence we need to add two guarded literals).

An example of possible frontiers in the reduced meet semi-lattice in Fig. 5.2 can be: the set $\{\{f_1\}, \{f_2\}\}$ or the set $\{\{f_3\}\}$.

The procedure described in this section is done at the *preprocessing stage*, once for each library function, and the resulting lattices can be used in verification of multiple programs as we discuss next.

### 5.3.2 Refinement of programs with library functions

The CEGAR approach we suggest here begins with verifying a first-order formula and iteratively refining the definition of library functions in the model till

eliminating all spurious counterexamples. We refine by loading more summaries (each of which represents a guarded literal), and this requires altering the original CEGAR approach. We change the refinement loop in the algorithm presented in Fig. 2.14 and add the usage of lattices of guarded literals, as shown in Fig. 5.3.

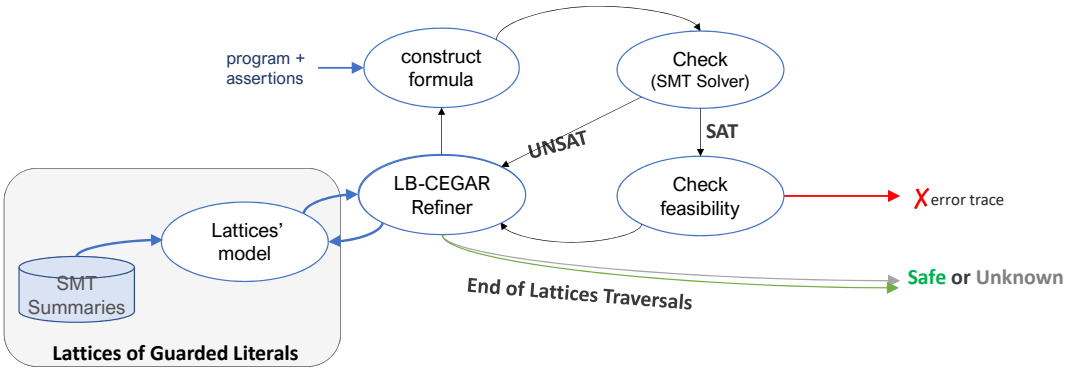


**Figure 5.3:** The modified CEGAR loop for the lattice-based counterexample-guided abstraction refinement in bounded model checking (LB-CEGAR-BMC) approach.

The **modified** refinement loop in LB-CEGAR-BMC approach contains the following steps:

- **Construct** a first-order formula as before, while encoding any library function as we encode uninterpreted functions (Fig. 5.3, *construct a formula* stage during the first iteration of the refinement loop).
- **Verify** the current first-order formula as before (Fig. 5.3, *check* stage).
- **Employ a feasibility check** as before, while using a theory under which the library functions in the current verification problem are fully defined and can be fully encoded (Fig. 5.3, *check feasibility* stage).

- **Terminate the refinement** once a counterexample passed the feasibility check (unsafe), the component of the *lattices of guarded literals* allows us to conclude that $t$ holds in $P$ (safe), or none of the lattices' literals can refine a counterexample that failed the feasibility check (unknown). That is, the termination depends now on both: the lattices state and the result from solve (Fig. 5.3, reporting *Error trace* in red colour, *SAFE* in green colour, or *Unknown* in grey colour).

- **Continue the refinement** as long as spurious counterexamples still exist, and there are additional guarded literals to refine with the encoding of the problem (Fig. 5.3, *LB-CEGAR refiner* stage). We construct a refined first-order formula with additional guarded literals from the lattices of guarded literals component (Fig. 5.3, *construct a formula* stage).

We can conclude that $t$ holds in $P$ with lattices of guarded literals by using their frontier (see Sec. 5.3.1). We decide which literals to add during each refinement by traversing the lattices at a certain order. We suggest an order of traversing lattices in Sec. 5.5.2. We later generalise the lattice traversal mechanism in Chap. 6.

## 5.4 Lattice Construction

In this section, we give additional definitions of a lattice of guarded literals characteristics and its construction algorithm from a set of equations and inequalities of properties of a library function. The construction algorithm outputs a lattice for a single library function given its set of properties. The inner function calls used in the construction algorithm are explained at the end of Sec. 5.4.2.

We note that while the size of the lattice can be exponential in the number of

expressions, the construction of the lattice is done as a *preprocessing step* once, and the results are used for verification of all programs with this function.

### 5.4.1 Definitions

A lattice of guarded literals $L$ contains **elements**, where each element contains a subset of guarded literals as we discuss in Sec. 5.3.1. The *guard of an element* (or its *assume* statement) is a conjunction of all the guards of its literals. We define the lattice's characteristic required for understanding our algorithms.

An element $E \in L$ is a **contradictory element** when $\bigwedge_{l \in E} l = false$ holds, where $l$ is a guarded literal as described in Sec. 5.3.1. No contradictory elements exist in $L$, as the lattice $L$ must contain only non-contradictory subsets of guarded literals in an element. Two elements $E, E' \in L$ are **logically equivalent elements** if $\bigwedge_{l \in E} l \iff \bigwedge_{l' \in E'} l'$ holds. We keep only one element in $L$ per subposet of logically equivalent elements.

**Definition 1** (Overlapping Guards)**.** *We say that two elements $E, E' \in L$ have overlapping guards when*

$$\bigwedge_{l \in E} assume(G) \wedge \bigwedge_{l' \in E'} assume(G') = true$$

*holds and the expression is in the following form: $l \stackrel{def}{=} assume(G) \wedge literal(x)$ where $G$ is a guard and $x$ is an equality or inequality of a property of function $f$.*

Note that, we keep the same partial order between elements in $L$ as in $SL(F)$ (for $L = \langle SL(F), \sqcap \rangle$) except for few changes for optimizations defined by the construction algorithm we present in Sec. 5.4.

### 5.4.2 Algorithm

The construction of a meet semi-lattice of guarded literals $L$ for a library function $f$ given a set of equations and inequalities of properties $F$, is described in Alg. 1. The algorithm consists of five main components:

**Construct a subset lattice from the input.** Given a set of statements (equation or inequality), we construct a set of guarded literals $F$ (line 1). Then given $F$, we construct a subset lattice $SL(F)$ (line 2).

**Consistency check.** For every element in $SL(F)$, we analyse the subset of guarded literals corresponding to this element (lines 3-10); all non-contradictory elements (lines 6-7) are added to the meet semi-lattice $L$ (line 8).

**Equivalence check.** Remove logical equivalent elements from $L$ (lines 11-20). We only keep the most upper element if it has equivalent elements.

**Cleanup.** After the execution of the checks and the removal of elements above, it is possible that in the resulting structure, an element has a single predecessor (lines 21-25). In this case, we unify the element with its predecessor (line 23). This process is repeated iteratively until all elements have more than one predecessor, except for the direct successors of the $\perp$ element.

**Fix Overlapping Assume Statements.** Strengthen an assumption to avoid overlapping between elements of the same predecessor (line 26, *fixG*); an overlapping *assume* statement is defined in Def. 1, Sec. 5.4.1.

The result of the algorithm is the meet semi-lattice $L$.

The exact $L$ depends on the input set of statements, as well as on the theory. We note, however, that $L$ can be used by the SMT solver with a different theory

than the one in which it was constructed, as long as an encoding of the guarded literals in SMT-LIB2 format with this logic exists. For example, the reduced meet semi-lattice in Fig. 5.2 can be used in EUF, even when its construction is done via propositional logic, since the encoding of $f_1, f_2$, and $f_3$ exists in EUF.

**Algorithm 1:** Lattice Construction

**Input :** $propertiesSet = \{(Y_1, y_1), \ldots, (Y_n, y_n)\}$: a set of pairs of equation or inequality of properties of $f$ with their assumptions.
**Output:** $L$

$F \leftarrow \bigcup_{(Y,y)\in propertiesSet}\{assume(Y) \wedge y, assume(\neg Y) \wedge \mathbf{true}\}$
$SL(F) \leftarrow buildSL(F)$
**foreach** *element* $E \in SL(F)$ **do**
    $minimise(E)$ //remove expressions that are generalised by other expressions in E
    $Query \leftarrow \bigwedge_{l\in E} l$
    $\langle result, \_\rangle \leftarrow checkSAT(Query)$
    **if** *result is* **SAT then**
        Add $E$ to $L$
    **end**
**end**
**foreach** *two elements* $E_{lower}, E_{upper} \in L$ *such that* $E_{lower} \subseteq_{SL(F)} E_{upper}$ **do**
    $Query \leftarrow \neg(\bigwedge_{l\in E_{lower}} l \iff \bigwedge_{l\in E_{upper}} l)$
    $\langle result, \_\rangle \leftarrow checkSAT(Query)$
    **if** *result is* **UNSAT then**
        **if** $\#E_{lower} < \#E_{upper}$ **then**
            $swap(E_{upper}, E_{lower})$
        **end**
        Remove $E_{lower}$ from $L$
    **end**
**end**
**foreach** *element* $E \in L$ **do**
    **if** $(\#immediateUpper(E)$ *is* $1) \wedge (\#immediateLower(immediateUpper(E))$ *is* $1)$ **then**
        Remove $E$ from $L$
    **end**
**end**
$L \leftarrow fixG(L)$ // Strengthen overlapping assumptions of all elements' successors
**return** $L$

### Overlapping assume statements

Elements in $L$ can have overlapping assume statements (Def. 1). In Fig. 5.2 for example, the $assume(Y)$ statement of $f_2$ originally was $(n > 0)$ thus the $assume$ statements of $f_1$ and $f_2$ overlap over many values, e.g., when $a = n$. In this

example (Fig. 5.2), we fix the *assume* statement of $f_2$ to avoid such overlapping by invoking *fixG* sub-procedure.

For each meet element $E \in L$, the sub-procedure *fixG*$(L)$ changes the assumptions of $E$'s immediate successors to remove any overlapping assumptions (see Def. 1). The *assume* statement of an immediate successor with a literal equal to *true* is updated by intersecting with negation of all other *assume* statements of the rest of the immediate successors of $E$, when removing any successor with an (altered) *assume* statement equals to $false$. An *assume* statement of an immediate successor is strengthen by intersecting with the negation of an *assume* statement of overlapping elements.

**A list of sub-procedures of main construction algorithm**

Let $f$ be a library function, $F$ be its set of guarded literals ($F$ is a finite set), $l, l', l''$ be guarded literals in $F$, $SL(F)$ be a subset lattice, $L$ be the lattice $L = \langle SL(F), \sqcap \rangle$, $E, E'$ be elements in a lattice, and $\psi$ be a logical formula, the algorithm above (Alg. 1) invokes the following procedures:

- $\#E$ is the number of guarded literals in $E$.
- *buildSL*$(F)$ constructs a *subset lattice* $SL(F)$ given $F$.
- *minimise*$(E)$, given an element $E \in L$, removes all literals in a subset of guarded literals *rem* such as that $rem \subset E$ and $(\bigwedge_{l \in E - rem} l) \iff (\bigwedge_{l \in E} l)$.
- *checkSAT*$(\psi)$ determines the satisfiability of $\psi$.
- *swap*$(E, E')$ swaps the current subset of guarded literals between $E$ and $E'$, while (roughly speaking) each element keeps its own edges.
- *immediateLower*$(E)$ gets all immediate predecessors of the element $E$.

- *immediateUpper*$(E)$ gets all immediate successors of the element $E$.
- *fixG*$(L)$ alters *assume* statements of elements in $L$ (Def. 1).

### 5.4.3 Lattice properties

Given a set of guarded literals $F$ for a library function $f$, the subset lattice $SL(F)$ is constructed as defined in Sec. 2.1. The height of $SL(F)$ is $|F|+1$ by construction, and the width is bounded by the following lemma on the width of a subset lattice.

**Lemma 1.** *For a set $S$ of size $s$, let $SL(S)$ be the subset lattice of $S$. Then, the width of $SL(S)$ is bounded by $\binom{s}{\lfloor \frac{s}{2} \rfloor}$.*

*Proof.* The bound follows from Sperner's theorem [And87] that states that the width of the inclusion order on a power set is $\binom{s}{\lfloor \frac{s}{2} \rfloor}$. □

These bounds also hold for $L = \langle SL(F), \sqcap \rangle$, since $L$ is a reduce lattice of $SL(F)$.

In Ex. 7, we use the subset of elements, $\{\{f_3\}\}$, which is the lattice frontier, instead of all elements of the lattice to prove the correctness of the code in Fig. 5.1. We can omit the rest of the lattice elements since the lattice frontier captures the whole input domain of the function. Informally, the claim follows from the structure of the subset lattice and the fact that the bottom element of lattice covers the whole domain. The claim and its formal proof are as follows.

**Claim 1.** *If the union of guards of a given set of guarded literals $S$ covers the whole domain of the input, then for each frontier $X(L_S)$ of the subset lattice $L_S$ of $S$, the union of guards of $X(L_S)$ also covers the whole domain of the input. And conversely, if the union of guards of a subset $X(L_S)$ of the elements of $L_S$ covers the whole domain of the inputs, then $X(L_S)$ is a frontier of $L_S$.*

*Proof.* We prove by induction on a subset lattice $L_S$ that any element $E \in L_S$ its guards refer to the same domain as the union of guards of all immediate successors of element $E$.

(base) The element $\emptyset$ has no guards and thus refers to the whole input domain $\mathbb{D}_{in}$. From the definition of subset lattice (Sec. 2.1.2), we know that each immediate successor of $\emptyset$ is a set of a single element from $S$. Since we added elements to $S$ to cover the missing subdomains (Sec. 5.3.1), we know that the union of all the guards of all items in $S$ captures all values in $\mathbb{D}_{in}$.

(step) For each element $E \in L_S$, the union of guards of all successors of $E$ is equivalent to the guard of $E$. Since $L_S$ is a subset lattice, then all immediate upper elements of an element $E \in L_S$ contain exactly one additional guarded literal from $S$. Since we added elements to $S$ to cover the missing subdomains, we know that any guarded literal has a guarded literal (or guarded literals) with an opposite guard in $S$, thus the union of any such pair of guards of these guarded literals leaves the original guard of $E$ the same; since each of the successor of $E$ must contain either the original guarded literal or its complementary guarded literal (or guarded literals). Therefore we get that the guard of the union of the successors of $E$ stays the same as required.

We use the above to prove the first part of the claim. We assume that the claim holds only if the union of guards of a given set of guarded literals covers the whole domain of the input, hence $\emptyset$ refers to the whole domain $\mathbb{D}_{in}$; since all chains start from $\emptyset$, and since the guard of an element is a union of guards of its immediate successors as proved by induction above, then if there is a frontier $X(L_S)$ where the union of all guards of all the guarded literals in $X(L_S)$ is not $\mathbb{D}_{in}$, then there is a chain from $\emptyset$ to maximal element without an element in $X(L_S)$, which contradicts the definition of a frontier (Sec. 5.3.1).

The LB-CEGAR-BMC algorithm (Alg. 2) and the LB-CEGAR algorithm (Alg. 5) can use a *reduced lattice* of a subset lattice $L_S$ (Sec. 5.4.2) by applying the following changes to the original structure: (1) removes elements with a guard equal to $false$ (i.e., consistency check), (2) removes logically equivalent elements by keeping a single element of these (i.e., equivalence check), (3) removes elements with a single successor where both have the same guard (i.e., Cleanup), and (4) fixes overlapping guard.

We fix overlapping guards while keeping the union of guards of all the successor of an element the same; thus the union of guards stays the same also in a frontier and therefore refers to the whole domain as before.

The rest of the changes of elements do not affect the union of guards; consistency check removes elements with no contribution to the input domain (as this equivalent to false), equivalence check affects only the number of possible frontiers, and cleanup removes elements with the same guard with a weaker expression compared to their single immediate successor.

The second part of the claim is trivial since any frontier of $L_S$ is a subposet of $L_S$. □

## 5.5 Lattice-Based Bounded Model Checking

In this section, we describe the lattice-based counterexample-guided abstraction refinement in bounded model checking (LB-CEGAR-BMC) algorithm for verifying programs with respect to a safety property and suggest an efficient lattice traversal algorithm for this version of the algorithm. We present a generalised version of this algorithm in Chap. 6.

### 5.5.1 Definitions

The lattices of guarded literals component in the LB-CEGAR-BMC's refinement loop (Fig. 5.3) contains sets of lattices for the model checking task. We describe in this section this component and its limitations.

For a program $P$ and a safety property $t$ such as that $P \cup \{t\}$ has functions which are missing the full definition in the current level of abstraction, we denote the set of all such functions in $P \cup \{t\}$ as $\{f_1, \ldots, f_m\}$. Each function $f$ has a meet semi-lattice $L = \langle SL(F), \sqcap \rangle$ ($F$ is a set of guarded literals of $f$). The set of all meet semi-lattices of the functions in $P \cup \{t\}$ is $\mathcal{L} = \{L_1, \ldots, L_m\}$.

We allocate a ***copy*** of $L$ per occurrences of $f$ in $P \cup \{t\}$. The LB-CEGAR-BMC uses these copies during verification process to be able to extract a different frontier per occurrences of $f$ in $P \cup \{t\}$. By using several copies, we can traverse lattices independently from other statements and hence can keep the queries to the SMT-solver much smaller. Using a single lattice for all occurrences of the function $f$ instead could damage the performance of the LB-CEGAR-BMC algorithm. We denote the ***set of all copies*** as $Lattices = \{L_{1,1}, \ldots, L_{m,k_m}\}$, where the lattice $L_{i,j}$ is the copy of $j$th occurrence of $f_i$ in $P \cup \{t\}$. We define $K$ to be the ***set of counters***, $K = \{k_1, \ldots k_m\}$ where $k_i \in K$ is the number of function calls of $f_i$ in $P \cup \{t\}$.

We assume each statement in the program under test has at most one occurrence of a function for the simplicity of our algorithm's description. If there is more than one library function in a statement, one can write an equivalent code that guarantees this property.

### 5.5.2 Traversal simulation of copies of lattices

The LB-CEGAR-BMC algorithm uses lattices' copies for a DFS style lattice traversal. We simulate the traversal separately for each occurrence of a function $f$ in $P \cup \{t\}$ that has a lattice in $\mathcal{L}$ by using the idea of having several copies of the lattice (one per occurrence of the function). As the lattice traversal idea might not be trivial, we outline in detail four common cases (that is, single copy with SAT and UNSAT results and two copies with SAT and UNSAT results) in the hope it will clarify the general idea, before outlining the LB-CEGAR-BMC algorithm in the next subsection. We refer in the text below to the specific line each step performed in the LB-CEGAR-BMC, which can be helpful in understanding how the simulation interacts with the general flow of the CEGAR loop; the simulation algorithm is part of the LB-CEGAR Refiner component (Fig. 5.3), where Alg. 3 describes the invocation of LB-CEGAR-Refiner after UNSAT result, Alg. 4 describes it after SAT result, and Alg. 2 describes the modified CEGAR loop and invokes the traversal simulation for both cases (after SAT and UNSAT result).

The basic idea of traversing a lattice is simple. We begin with the weakest subset of guarded literals of the lattice. In the main refinement loop, we gradually move to upper elements of the lattice containing a stronger subset of guarded literals according to the counterexample presented to us as shown schematically in Fig. 5.3. However, the exact order of traversing a lattice depends on the order of the statements, the inner partially order of each lattice, the order we traverse the copies (for example, which copy the refiner tries first, the counterexample received from the SMT solver and as well our decision how and if to refine it), the theory we use to encode our initial problem, and other heuristics.

In this section, we present a traversal simulation of a copy (of a lattice) based on the lattice structure and the order of statements in code. We describe in the implementation part (Sec. 5.6.1) heuristics on the order of the statements in the context of bounded model checking. We use the following symbols though the section; let $f$ be a library function, $L$ a lattice for $f$, $c$ a copy of $L$, $E$ an element, $X(c)$ the frontier of $c$, $P$ a program with occurrences of $f$, $t$ a safety property in the specification of $P$, $\hat{\varphi}$ a first-order formula, *CEX* a counterexample, and $\chi$ a first-order formula of currently used properties.

### Traversal of a single copy of a lattice

A lattice traversal on a copy $c$ of a lattice $L$ for a function $f$ (for an occurrence of $f$ in $P \cup \{t\}$) starts with $\bot$ element (that is, $\emptyset$), adding no guarded literals to the query $\hat{\varphi}$, which models $P$ with the negation of $t$ (Alg. 2, line 5). The traversal on $c$ continues according to the last result from the solver (Alg. 2, line 7), which is either SAT or UNSAT result. If the solver returns 'unknown', we treat it as a **SAT** result. We describe each of the cases in detail, starting with the **SAT** result with a single copy, $c$.

During execution, each time $\hat{\varphi} \wedge \chi$ is **SAT**, we traverse to an upper element of $c$ which eliminates the counterexample, *CEX*, as long as *CEX* is a spurious counterexample (Alg. 4, lines 3-9). Once a real counterexample is obtained, the algorithm terminates and returns **Unsafe** with the counterexample, *CEX*. Reaching a maximal element in $c$ during the traversal indicates that the guarded literals in $L$ cannot refine this occurrence of $f$ in $P \cup \{t\}$, which also terminates the refinement in the LB-CEGAR-BMC algorithm (Alg. 2) and returns **Unsafe** (but with no counterexample).

In Fig. 5.4, we sketch the flow of the case of a **SAT** result from the solver. The

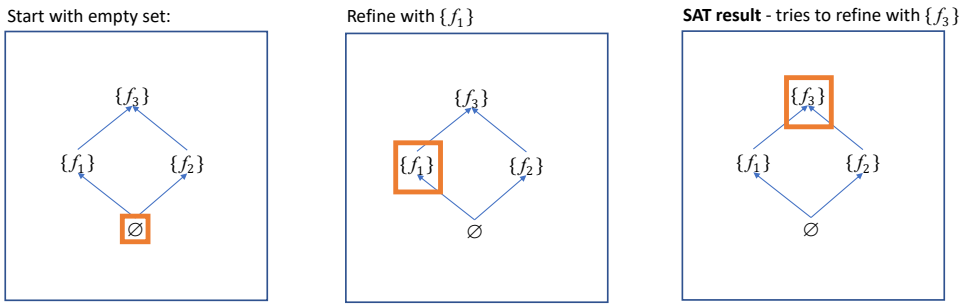


**Figure 5.4:** Lattice traversal: single lattice, single copy, with last result **SAT**.

orange square marks the current element during each of the following steps. The traversal simulation: (*left*) tries solving the problem with no additional guarded literals and receives **SAT** result from the solver, and traverses to an upper element (where we assume $\{f_1\}$ refines the current counterexample), (*middle*) adds $\{f_1\}$ expressions to the query $\hat{\varphi}$ (in Alg. 2, line 5-6), receives **SAT** (in Alg. 2, line 7), and assuming none of the $if$ conditions in Alg. 2 lines 15 and 18 holds, uses a stronger subset of expressions $\{f_3\}$. (*right*) $\{f_3\}$ is a maximal element, in this example if the query $\hat{\varphi}$ with $\{f_3\}$ is **SAT**, the refinement fails and the LB-CEGAR-BMC algorithm returns **Unsafe** (Alg. 2, line 19).

Once the query $\hat{\varphi}$ with additional guarded literals (that is the subset of guarded literals of an element $E \in c$) is **UNSAT**, the traversal of a single copy skips the successors of $E$, adds $E$ to $X(c)$ (the frontier of $c$), and continues the traversal with one of the siblings of $E$ according to the DFS order from left to right. The traversal simulation of $c$ terminates, if there are no remaining siblings of $E$, and outputs **Safe**.

In Fig. 5.5, we sketch the flow of the case of a **UNSAT** result from the solver. The orange square marks the current element during each of the following steps.

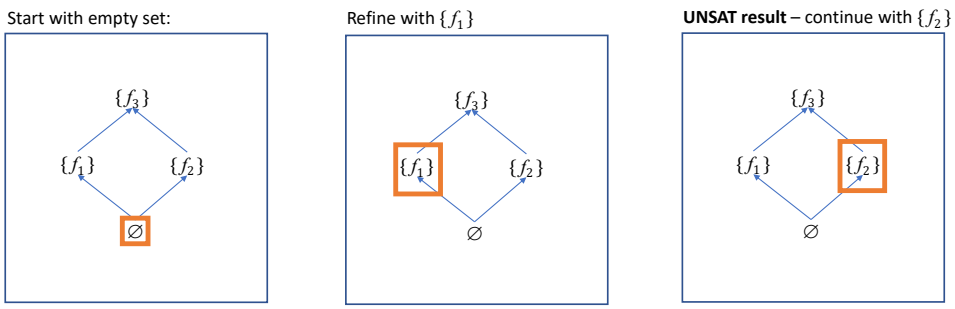


**Figure 5.5:** Lattice traversal: single lattice, single copy, with last result **UNSAT**.

(*left*) We assume the scenario at the beginning of the refinement is the same (that is, we require some guarded literals to refine $\hat{\varphi}$). The traversal simulation after it: (*middle*) adds $\{f_1\}$ expressions to the query $\hat{\varphi}$ (in Alg. 2, line 5-6), receives **UNSAT** (in Alg. 2, line 7), adds $\{f_1\}$ to the frontier of $c$ (Alg. 3, line 2), and (*right*) uses a different subset of expressions $\{f_2\}$ for a different input domain of the function call we refine (Alg. 3, line 7).

Note that, in this example, $\{f_3\}$ is the next element from $\{f_1\}$ according to the DFS order. However, since $\{f_1\}$ is sufficient to prove **UNSAT**, there is no need to strengthen the expressions in $\{f_1\}$ and use $\{f_3\}$. Therefore, we can skip $\{f_3\}$ in the DFS order and continue with $\{f_2\}$.

### Traversal of multiple copies of a lattice

We use the description of a single copy traversal when describing the traversal simulation of multiple copies. Each of the copies follows the mechanism of a single copy traversal while using additional guarded literals. These can be guarded literals either of an element or a frontier, of other copies (Alg. 2, line 5, or Alg. 4, line 5).

As long as the result from the solver is **SAT** (Alg. 4, line 6, or Alg. 2, line 7), the refinement continues the traversal simulation of multiple copies with no valid frontier either by strengthening the subset of guarded literals of a copy (Alg. 4, line 6) or by trying to add guarded literals from other copies (Alg. 4, next iteration in the *for* loop in line 1).

We follow the same traversal simulation mechanism of a single copy per copy while searching for such a subset of guarded literals (Alg. 4, lines 3-9). We continue the refinement if one of the copies refines the current counterexample *CEX* (Alg. 4, line 8, returns *true*); in that case, we return to the main loop of the LB-CEGAR-BMC algorithm (Alg. 2, line 18, and then skips to the next iteration of the main loop, lines 4-22). A subset of guarded literals refines *CEX* if with this subset *CEX* is not feasible (Alg. 4, line 6, returns **UNSAT**). However, if none of the copies in *Lattices* can refine the current counterexample *CEX*, the refinement fails (Alg. 4, line 13, returns $false$).

In Fig. 5.6, we illustrate these two cases with two copies of $L$. We use either copy $c$ or other copies to refine an occurrence of $f$ (where $c$ is a copy of $L$ for this occurrence of $f$). $c$ is "copy 1" and is the current copy the algorithm constructs its frontier. "copy 2" is the other copy and has no frontier yet. The orange square marks the current element in each of the copies during each of the following steps.

The traversal simulation for "copy 1" and "copy 2": (*top-left*) tries solving the problem with no additional guarded literals and receives **SAT** result from the solver, and traverses to an upper element in "copy 1" (where we assume $\{f_1\}$ refines the current counterexample). (*top-right*) We then return to the main loop in the LB-CEGAR-BMC algorithm (Alg. 2, line 18), continue to the next iteration, and try solving the problem with $\{f_1\}$ of "copy 1". (*bot-left*) The traversal simulation continues and traverses to an upper element in "copy 1" (where we as-

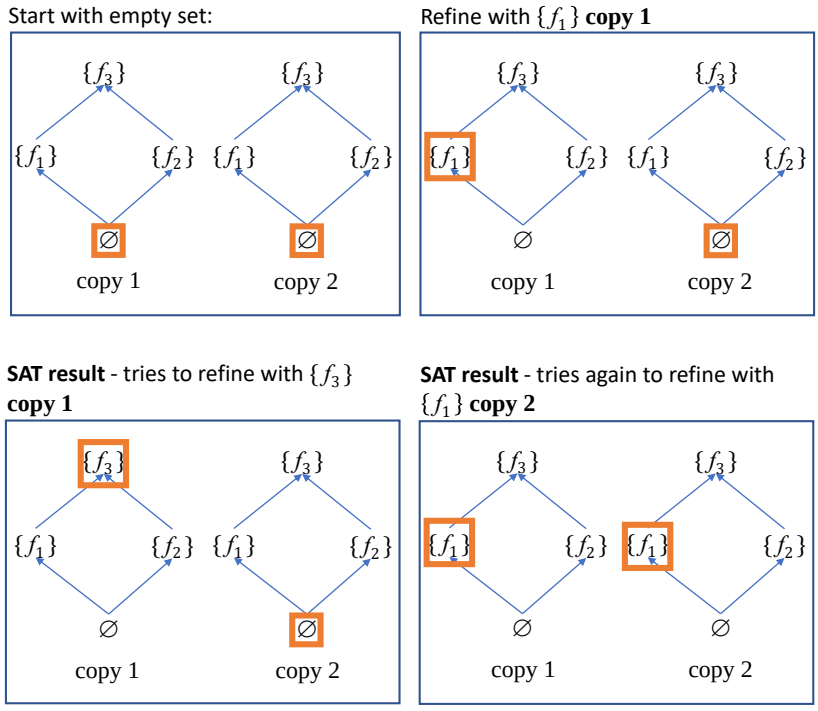


**Figure 5.6:** Lattice traversal: single lattice, two copies, with last result **SAT**.

sume $\{f_3\}$ does not refine the current counterexample and is a maximal element of "copy 1"), (*bot-right*) resets back to $\{f_1\}$ in "copy 1", and traverses to an upper element in "copy 2".

Following an **UNSAT** result, the algorithm either updates the frontier of the current copy (Alg. 2, line 12) and (once the condition in Alg. 3 line 3 holds) starts the traversal of another copy (Alg. 3, line 5) or, terminates (Alg. 2 line 9). In the latter case, the algorithm terminates once using the frontiers of all the copies is sufficient to prove that $P$ is safe with respect to a property $t$. In the former case, after the LB-CEGAR-BMC algorithm (Alg. 2) finds a frontier, it uses this frontier to construct a first-order formula adding all of its guarded literals (Alg. 2, lines 5-6, and Alg. 4, lines 5-6). Hence, we use the guarded literals in the frontier for extracting frontiers for the rest of the copies. There is no use of a current

element of copy $c$ once its traversal ended, we then use only the guarded literals in its frontier.

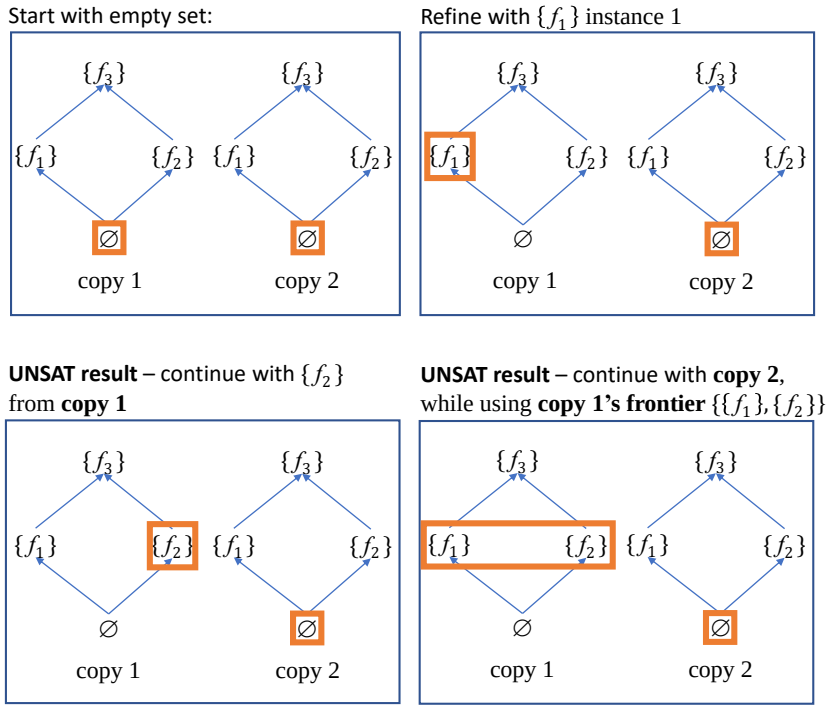


**Figure 5.7:** Lattice traversal: single lattice, two copies, with last result **UNSAT**.

We illustrate this case in Fig. 5.7 for two copies of a lattice (starting upper-left side and left to right). The algorithm extracts and uses a frontier of the first copy '("copy 1") to assist refine the occurrence of function f of the second copy ("copy 2"). The orange square marks the current element or the copy's frontier in each of the copies during each of the following steps.

The traversal simulation for "copy 1" and "copy 2": (*top-left*) tries solving the problem with no additional guarded literals and receives **SAT** result from the solver, and traverses to an upper element in "copy 1" (where we assume $\{f_1\}$ refines the current counterexample). (*top-right*) We then return to the main loop in the LB-CEGAR-BMC algorithm (Alg. 2, line 18), continue to the next iteration,

solve the problem with $\{f_1\}$ of "copy 1" (where we assume $\{f_1\}$ refines the current sub-domain of values), and add $\{f_1\}$ to the frontier of "copy 1" (Alg. 2, line 12). (*bot-left*) The traversal simulation continues and traverses to a sibling element of $\{f_1\}$ in "copy 1", which is $\{f_2\}$ in "copy 1". (*bot-right*) We then return to the main loop in the LB-CEGAR-BMC algorithm (Alg. 2, line 18), continue to the next iteration, solve the problem with $\{f_2\}$ of "copy 1" (where we assume $\{f_2\}$ refines the current sub-domain of values), and add $\{f_2\}$ to the frontier of "copy 1" (Alg. 2, line 12). We then have a valid frontier of "copy 1", $\{\{f_1\}, \{f_2\}\}$, which we use when extracting the frontier of "copy 2".

In Chap. 6, we present a generalised version of the algorithm in Sec. 5.5.3 and describe a set of other heuristics to traverse lattices generalising the ideas here. We then prove that the generalised version of the traversal simulation terminates, its complexity, and that if the refinement outputs a positive result (that is, the program is safe with respect to the given bound), then there are no counterexamples up to the given depth in the program (that is, in its loop-free version). We also present in Chap. 6 a set heuristics to achieve optimal order of traversing the copies via experimental evidences.

### 5.5.3 Algorithm

The *lattice-based counterexample-guided abstraction refinement in bounded model checking* algorithm (LB-CEGAR-BMC) for verifying programs is described in Alg. 2 and two additional LB-CEGAR-BMC algorithm's sub-procedures: Alg. 3 and Alg. 4. The rest of LB-CEGAR-BMC algorithm's sub-procedures are described at the end of this section shortly.

The LB-CEGAR-BMC algorithm (Alg. 2) takes the symbolically encoded program $P$ with a safety property $t$ and constructs an over-approximating formula $\hat{\varphi}$

of the problem in a given logic (line 1). The LB-CEGAR-BMC algorithm (Alg. 2) refines $\hat{\varphi}$ by adding and removing guarded literals from lattices in $\mathcal{L}$ according to the traversal on their copies, where we have a copy of a lattice per refined expression (main loop, lines 4-22). The algorithm terminates once it has proved the current $\hat{\varphi}$ is **Safe** (lines 9-11), after extracting a real counterexample (lines 15-17), or after using all available guarded literals in all lattices' copies while still receiving spurious counterexamples (lines 18-20 or 24). The exact order, in which we add guarded literals to eliminate counterexamples, is arbitrary at this point, except for the partial order induced by the copies and their statements in the code. We discuss the partial order in Alg. 4, and we offer several heuristics in Chap. 6.

The refinement in the LB-CEGAR-BMC algorithm (Alg. 2) is finite and returns **Unsafe** if $t$ does not hold in $P$. A counterexample in the last known precision is returned when $t$ does not hold in $P$ and the guarded literals in the lattices' copies refine the over-approximate functions in $\hat{\varphi}$. The LB-CEGAR-BMC algorithm (Alg. 2) returns **Safe** if and only if the guarded literals in the lattices' copies refine the over-approximate functions in $\hat{\varphi}$ and $t$ holds in $P$. The lattices we use in the LB-CEGAR-BMC algorithm (Alg. 2), are described in the previous sections in this chapter. Their description is essential for understanding the LB-CEGAR-BMC algorithm (Alg. 2), mainly lines 1-4 in Alg. 2.

### Frontiers construction

The LB-CEGAR-BMC's sub-procedure: *updateFrontier* (Alg. 3) adds the current element in the current copy, $c$, to its frontier (line 2) and continues the traversal on the lattice's copy via the sub-procedure $traverse_{UNSAT}$ (line 7). If the traversal

**Algorithm 2:** LB-CEGAR-BMC

**Input :** Program $P = \{s_1 \coloneqq (x_1 = t_1), \ldots, s_n \coloneqq (x_n = t_n)\}$,
Safety property $t$,
Semi-lattices' set $\mathcal{L} = \{L_1, \ldots, L_m\}$.
**Output:** $\langle$**Safe**, $\bot\rangle$ or $\langle$**Unsafe**, $CEX\rangle$ or $\langle$**Unsafe**, $\bot\rangle$

1 $\hat{\varphi} \leftarrow \bigwedge_{s \in P \cup \{t\}} convert(s)$
2 $K \leftarrow getFuncCallsCounts(\mathcal{L}, (P \cup \{t\}))$ // Construct the set $K$
3 $Lattices \leftarrow \bigcup_{i=1}^{m} \bigcup_{j=1}^{k_i \in K} (L_{i,j} \leftarrow initialiseLI(s, L_i))$
4 **while** ($\exists c \in Lattices$ : *element*($c$) *is not a maximal element)* **do**
5 $\quad \chi \leftarrow \bigwedge_{(c \in Lattices)} currentFacts(c)$
6 $\quad Query \leftarrow \hat{\varphi} \wedge \chi$
7 $\quad \langle result, CEX\rangle \leftarrow checkSAT(Query)$
8 $\quad$ **if** *result is* **UNSAT then**
9 $\quad\quad$ **if** ($\chi$ *is* **true**) $\vee$ ($\forall c \in Lattices$ : $hasFrontier(c)$) **then**
10 $\quad\quad\quad$ **return** $\langle$**Safe**, $\bot\rangle$ //Safe - Quit
11 $\quad\quad$ **end**
12 $\quad\quad Lattices \leftarrow updateFrontier(Lattices)$ //Continue the Traversal
13 $\quad$ **end**
14 $\quad$ **else**
15 $\quad\quad$ **if** $checkRealCE(Query, CEX)$ **then**
16 $\quad\quad\quad$ **return** $\langle$**Unsafe**, $CEX\rangle$ //Real counterexample - Quit
17 $\quad\quad$ **end**
18 $\quad\quad$ **if** $!refineCEX(P, t, Query, CEX, Lattices)$ **then**
19 $\quad\quad\quad$ **return** $\langle$**Unsafe**, $\bot\rangle$ //Cannot Refine - Quit
20 $\quad\quad$ **end**
21 $\quad$ **end**
22 **end**
23 // End Of Main Loop
24 **return** $\langle$**Unsafe**, $\bot\rangle$ //Cannot refine - Quit

of $c$ ended (line 3), then the sub-procedure starts traversing a copy without yet a frontier (lines 3-6, and line 7).

Note that, (i) each copy has its own frontier (as we explain in Sec. 5.4.3 and Sec. 5.5.1), (ii) a copy's traversal ends when we either find a valid frontier or reach a maximal element, and (iii) this sub-procedure, *updateFrontier* (Alg. 3), keeps changing the same lattice's copy till either extracting a valid frontier for this copy or the main algorithm, the LB-CEGAR-BMC algorithm (Alg. 2), terminates.

**Algorithm 3:** *updateFrontier* (LB-CEGAR-BMC's sub-procedure)

**Input :** Lattices' copies *Lattices*
**Output:** Updated lattices' copies *Lattices*

```
1 c ← last changed copy in Lattices
2 Add element(c) to the frontier of c
3 if hasFrontier(c) then
4 |   ∀x ∈ Lattices.¬hasFrontier(x) ⟹ reset(x, ⊥)
5 |   c ← an item in {x|x ∈ Lattices ∧ ¬hasFrontier(x)}
6 end
7 traverse_UNSAT(c)
8 return Lattices // Returns back to the main loop in Alg. 2 line 11
```

### Abstraction refinement of a spurious counterexample

The LB-CEGAR-BMC's sub-procedure: *refineCEX* (Alg. 4) describes the refinement of a single spurious counterexample *CEX* via copies of lattices in *Lattices*. The order of applying the copies' guarded literals during refinement is according to (1) the order in the code statements with an occurrence of F (the list of heuristics is in Sec. 5.6.1), and (2) the partial order induced by each lattice. The rest of our choices are arbitrary in this chapter (we change it in the next chapter).

The main loop (lines 1-12) searches for a lattice's copy that refines a counterexample given as input (that is, *CEX*). The inner loop (lines 3-10) searches for an element in the current copy that its subset of guarded literals refines *CEX* (lines 3-10). If the inner loop reaches a maximal element (line 3), then the main loop skips this copy and continues the search with the next copy in *Lattices* (line 1). In that case, we also revert the changes in the copy that its traversal reached a maximal element (line 11). We do not include copies with a frontier as these are already part of the original query, *Query*.

A single counterexample refinement is successful if the query (line 6) is **UNSAT**, that is, it can detect now that this counterexample *CEX* is infeasible by using additional guarded literals (added in lines 4-5). The refinement fails if, for all copies in *Lattices*, no element can refine the current counterexample *CEX* (line

13). The refinement order is determined by the way the sub-procedure *refineCEX* (Alg. 4) goes over statements $s \in P \cup \{t\}$ (line 1), which is done according to sets of basic heuristics defined in Chap. 7. In the implementation section in this chapter (Sec. 5.6.1), we describe these heuristics shortly.

**Algorithm 4:** *refineCEX* (LB-CEGAR-BMC's sub-procedure)

```
   Input  : Program P = {s_1 ≔ (x_1 = t_1), ..., s_n ≔ (x_n = t_n)},
            Safety property t,
            Query Query,
            Counterexample CEX,
            All lattices' copies Lattices.
   Output: true or false
 1 for s ∈ P ∪ {t} with c ∈ Lattices and ¬hasFrontier(c) do
 2     n ← element(c) //To reset later to original location n
 3     while element(c) is not a maximal element do
 4         traverse_SAT(c) // Try to refine with a stronger subset
 5         χ ← currentFacts(c)
 6         ⟨result, _⟩ ← checkSAT(Query ∧ CEX ∧ χ)
 7         if result is UNSAT then
 8             return true // Refined CEX; return back to main loop in Alg. 2 line 17
 9         end
10     end
11     reset(c, n) // Cannot refine CEX with this copy, rest and continue to next copy
12 end
13 return false // Return to main loop in Alg. 2 lines 17-18
```

### Validating counterexamples

The LB-CEGAR-BMC algorithm (Alg. 2) checks if *CEX* is a spurious counterexample via *checkRealCE* sub-procedure (Alg. 2, line 14) similarly to the counterexample check in Chap. 7 and returns either true with a real counterexample when all queries are **SAT**, or false otherwise.

The solver produces an interpretation for the variables or a partial interpretation of uninterpreted functions and uninterpreted predicates in the case of EUF, for statements $s \in P \cup \{t\}$ in the current precision. The counterexample validation (*checkRealCE* sub-procedure) determines whether the conjunction of a statement

$s \in P \cup \{t\}$ and a counterexample *CEX* with an interpretation or partial interpretation is **UNSAT** in a more precise theory; an **UNSAT** result in any of the queries indicates that the counterexample is indeed spurious. A more precise theory can be the theory of bit-vectors as in Chap. 7 or the theory the meet semi-lattice was built with. If no available description of the function $g$ with the current query exist in a more precise theory, we assume *CEX* is spurious.

### A list of sub-procedures of the LB-CEGAR-BMC algorithm

We describe the rest of the function calls in general. Let $s$ be a statement in $P \cup \{t\}$, $\psi$ a logical formula, *CEX* a counterexample, $x$ a meet semi-lattice of a statement $s$ with a function $f$, and $x'$ a lattice's copy of $x$, the LB-CEGAR-BMC algorithm (Alg. 2) with its two sub-procedures *updateFrontier* (Alg. 3) and *refineCEX* (Alg. 4) invoke the following procedures:

- *checkRealCE*($\psi$, *CEX*) returns true if the formula *CEX* is a real counterexample of formula $\psi$.
- *checkSAT*($\psi$) determines the satisfiability of a formula $\psi$.
- *convert*($s$) creates a symbolic formula in the initial logic for statement $s$.
- *currentFacts*($x'$) retrieves the formula with the guarded literals in $x'$ which is a union of all elements in the frontier, an intersection of the guarded literals in the current element or, *true* if the current element is $\bot$.
- *element*($x'$) retrieves the current element in $x'$ or $\top$ for $x'$ with a full frontier.
- *hasFrontier*($x'$) returns *true* if $x'$ has a frontier.
- *initialiseLI*($s, x$) if there is a meet semi-lattice for *operation*($s$) in $\mathcal{L}$, creates a copy of a meet semi-lattice $x$ for $s$.

- *getFuncCallsCounts*$(\mathcal{L}, (P \cup \{t\}))$ gets counters of occurrences per function $f$ with a lattice in $\mathcal{L}$.
- *operation*$(s)$ retrieves the operation or function call name in $s$.
- *reset*$(x', n)$ sets the current element in the traversal simulation of $x'$ to be location of element $n$ and reverts the inner state of the search.
- *traverse*$_{SAT}(x')$ simulates a traversal of $x'$ from the current element to elements with stronger subset of guarded literals as described in Sec. 5.5.2.
- *traverse*$_{UNSAT}(x')$ simulates a traversal of $x'$ in DFS style as described in Sec. 5.5.2.

The sub-procedure *updateFrontier* (Alg. 3) and the sub-procedure *refineCEX* (Alg. 4) are not part of the description on the list above as both are described in detail in the Alg. 3 and Alg. 4. These two sub-procedures are called in the main loop of the LB-CEGAR-BMC algorithm (Alg. 2) in lines 12 and 18, respectively.

## 5.6 Implementation and Evaluation

In this section, we present the basic implementation of LB-CEGAR-BMC algorithm in HiFrog.

In the next chapter, we generalise the algorithm to be able to deal with much more complex cases. For the current implementation, we present an initial evaluation of our prototype on a single library function: the modulo operation with several occurrences in the code. We use the evaluation here to understand better the improvements required for dealing with complex cases. We present some of the improvements in the next chapter and we discuss the rest in our future work in Chap. 9.

### 5.6.1 Implementation of the LB-CEAGR-BMC algorithm

The LB-CEAGR-BMC algorithm was implemented on the SMT-based function summarisation bounded model checker HiFrog (Chap. 4, [AAC+17]) and used either the Z3 SMT solver [DMB08b] or the OpenSMT2 SMT solver [HMAS16]. The model checker was compiled with the GNU C++ compiler and the O3 optimization level. The preprocessing steps were implemented as a set of BASH scripts. The scripts for the construction of a meet semi-lattice, the meet semi-lattices for modulo operation, the complete experimental results, and the source code are available at [Git19a, Git19b, HiF19].

#### Preprocessing

**Extraction of guarded literals.** A preprocessing step of our framework was extracting a set of guarded literals $F$ for a library function $f$. The equations and inequalities of properties of $f$ can be imported from another program or a library and parsed into a set of guarded literals per library function. In the experimental results, we imported equations and inequalities from the Coq proof assistant [The19a] and Wikipedia [Mod19], where $f := mod$ (modulo function). We used a subset of lemmas, theorems and definitions of properties of the modulo function from [Mod19, The19a] as is, as the data is simple to use, well known, and reliable. We translated the equations and inequalities into the SMT-LIB2 format manually (see [HiF19] for the results of this translation).

**Validation.** The validation test is as follows. Given a function $f$, a set of expressions $F$, a statement $s$ such that $l$ ($l \in F$) is sufficient to verify $s$, test if $s \wedge l$ is **UNSAT**. A complementary validation test is the sanity check which verifies that the guarded literals in $F$ are not contradictory in the theory under

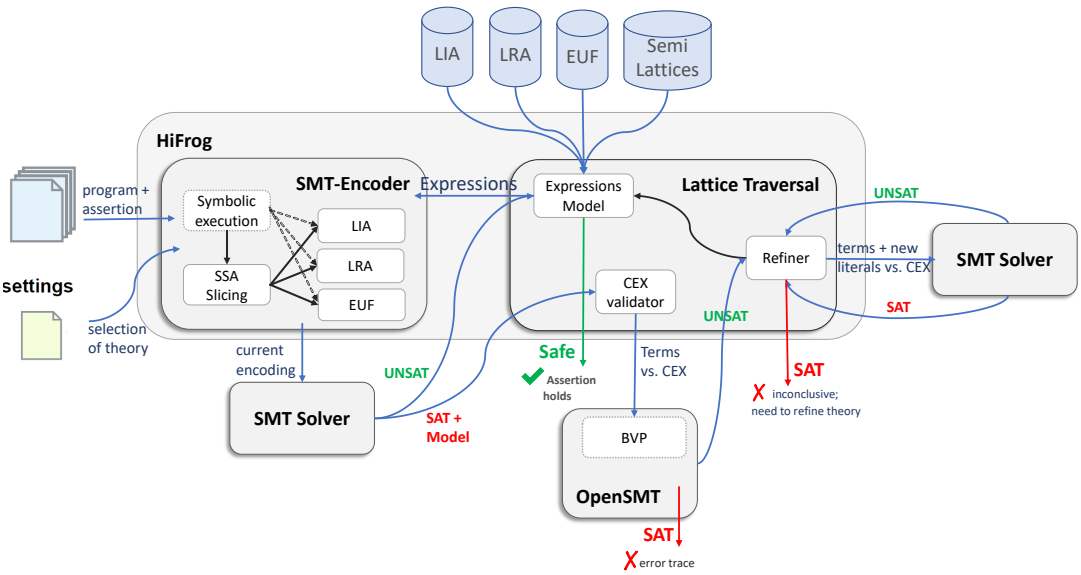


**Figure 5.8:** Architecture of LB-CEGAR-BMC implementation in SMT-based model checking framework.

which $f$ is fully defined. We describe in Sec. 5.6.2 the validation tests for the guarded literals of properties of the modulo function used in the evaluation of LB-CEGAR-BMC. The function $f$ was *mod* and the set of guarded literals was $F := F_{mod}$ with 31 guarded literals.

**Construction.** The script contained greedy optimisation of the construction of a meet semi-lattice algorithm (Alg. 1) to avoid, if possible, an exponential number of calls to the solver. The first loop (in lines 3-10) started from the smallest subsets of expressions (to largest), all successors of a contradictory element (see Sec. 5.4.1) were pruned, and the second loop (Alg. 1, lines 11-19) considered only pairs of (roughly speaking) connected elements.

### The LB-CEGAR-BMC algorithm

The overview of the interaction between HiFrog, the refiner in HiFrog and the SMT solvers is shown in Fig. 5.8. In the current prototype, we added guarded

literals of the meet semi-lattice as SMT summaries, while checking before applying a summary that its *assume* statement (the guard of a literal) holds for better performance. Any function that had no precise encoding in the current level of abstraction (Fig. 5.8, given as an input file of settings to HiFrog) was added as a candidate for refinement.

The spurious counterexample check was done via the CEX validator using bit-vector logic similar to the counterexample check in Chap. 7, Sec. 7.4.2 with one of the straightforward heuristics:

1. Forward order with a single refinement.
2. Backward order with a single refinement.
3. Forward order with simultaneous refinement.
4. Backward order with simultaneous refinement.
5. Forward order with dependency refinement.
6. Backward order with dependency refinement.
7. Forward order with simultaneous and dependency refinement.
8. Backward order with simultaneous and dependency refinement.

We refined a single statement each time we called the CEX validator by validating a single statement only with option (1) and option (2). We refined a block of statements with option (3) and option (4). We refined a statement with all the statements this statement required to compute the expression in this statement with option (5) and option (6). We refined a block of statements with all the statements this block required to compute the expression in any of the statements in this block with option (7) and option (8).

For all options, we started the check either from the first statement [3] forwards (1,3,5 or 7) or the assert statement backwards (2,4,6 or 8) and refined a statement or statements (according to the selected option) that failed the counterexample feasibility check via the CEX validator. We used only option 5[4] in our experiments (Sec. 5.6.2). However, all 8the options were implemented in HiFrog also for the LB-CEGAR-BMC algorithm.

The CEX validator output was either *true* with a real counterexample when all queries were SAT, or $false$ with a set of candidates to refine otherwise. Each query contained a statement (single refinement or simultaneous refinement), or a set of statements (dependency refinement) with an assignment of values taken from the current counterexample in SMT-LIB2 format.

The lattice traversal component contained three sub-components:

1. *The model.* This component contained the meet semi-lattices loaded to HiFrog and their copies per occurrence of a function we refined.

2. *CEX validator.* This component validated the counterexample and reported real counterexamples once found.

3. *Refiner.* This component did the refinement, added guarded literals to and removed guarded literals from the encoding of the problem (program $P$ with a safety property $t$). This component also interacted with the CEX validator and the rest of the system and terminated the refinement for each of the three possible cases described in the LB-CEGAR-BMC algorithm (Alg. 2).

The SMT solvers instances used equality logic with uninterpreted functions (EUF) or linear arithmetics with EUF for modelling and the OpenSMT2 SMT solver instances with fixed-width bit-vectors for CEX validation.

[3] That is, the entry point of the program under test.
[4] The default option in HiFrog, see Chap. 7, Sec. 7.4.2 for more details.

**Table 5.1:** Results of validation test.

| | | **Safe** | **Unsafe** |
|---|---|---|---|
| HiFrog+UDS | LIA | 37 | 46 |
| | EUF | 9 | 46 |
| HiFrog | Prop. | 20 | 44 |
| | LIA | 3 | 46 |
| | EUF | 2 | 46 |
| **Total** | | **45** | **46** |

### 5.6.2 Experimental results

This section describes the evaluation of the quality of meet semi-lattices of guarded literals and the evaluation of the LB-CEGAR-BMC approach. The experiments ran on a 64-bit virtual machine with Ubuntu 16.04 Linux system, single CPU and 10000 MB base memory. The time-out for all experiments was at 4000s, and the memory limit was 3GB.

#### Validation results for modulo function

In this section, we describe the validation results for the set of guarded literals used in the construction of meet semi-lattices of the modulo operation. The results here are not part of the experiments of any of the algorithms presented in the chapter. We did not use any of the algorithms in this chapter. The validation part is only for ensuring as proper input as possible, where the input for the construction of the lattice, is a set of these guarded literals as an SMT summary.

We built a set of benchmarks of 91 C-programs with a single assertion, with 45 safe and 46 unsafe instances for all the 31 guarded literals of properties of the modulo function (as shown in Table 5.1, last row). We translated into SMT-LIB2 format all the 31 guarded literals we used. The majority of them had at least one **UNSAT** and one **SAT** benchmark. We ran HiFrog with SMT

summaries such as that each guarded literal had its own SMT summary. We used the OpenSMT2 SMT solver for EUF encoding and the Z3 SMT solver for LIA with EUF encoding.

The column *Safe* in Table 5.1 contains the total number of solved instances with different SMT theories and with or without SMT summaries. It describes the result of the validation tests for three of the supported encoding modes of HiFrog (Chap. 4, [AAC+17]): propositional logic, LIA, and EUF encoding with summaries (HiFrog+UDS LIA, HiFrog+UDS EUF) and without summaries (HiFrogProp., HiFrogLIA, HiFrogEUF); propositional logic supports modulo function (hence, we have only five rows in Table 5.1 instead of six). We did not expect any of the safe-benchmarks to be solved correctly with EUF and LIA without SMT summaries. However, we added these results to Table 5.1 merely to check that a safe benchmark could not be solved without an additional guarded literal.

With propositional encoding we could verify 20 out of 45 safe benchmarks. The remaining 25 validation tests had time-out or out of memory issues or a different result than expected[5]. A different result was possible due to choice of implementation of the modulo operation in C (e.g., modulo of 0 or the inverse of operations of elementary arithmetic). The cause for resources problems was expensive operations at the bit-level precision in the crafted benchmarks. Neither LIA nor EUF can express the modulo function and thus, in general, could not terminate with a **Safe** result, as reported. However, we reported a few cases in which LIA and EUF terminated with a **Safe** result due to simplifications in the representation of the program in its Static Single Assignment (SSA) form (few examples of the SSA form of these validation tests are available at [Git19a, HiF19]).

[5]In our case, when the resource from which we loaded these properties disagree with C99.

The validation results of the properties of the modulo functions as user-defined summaries (UDS) with LIA (that is, combining a guarded literal's summary with the LIA in the encoding of the problem) verified 37 out of 45 safe benchmarks. The remaining 8 validation tests contained non-linear operators (e.g., $(a+b*c)\%c$) and thus could not be verified with linear arithmetic. The validation results of the properties of the modulo functions as user-defined summaries (UDS) with EUF reported only 9 instances as **Safe** (EUF's expected behaviour on mathematical benchmarks).

The last column of Table 5.1 investigates the unsafe benchmarks, corresponding to the sanity check. All expressions are non-contradictory hence the addition of guarded literals did not change the result of verification of the **Unsafe** instances for both EUF and LIA, with SMT summaries.

### Evaluation of LB-CEGAR-BMC

LB-CEGAR-BMC can solve instances with library functions, given a set of lattices for these functions; because our choice of a library function for evaluation (for the work in this chapter) was the modulo function, we did not expect it to outperform any other well-established methods but to obtain similar results. Since (1) there is efficient support in solvers for modulo function in propositional logic, and (2) the current LB-CEGAR-BMC's implementation (in this chapter) had naive heuristics on the statements' refinement order and no optimizations. And yet, we chose the modulo function as it (a) is commonly used in C code and tend to appear in benchmarks (like SV-COMP), (b) has a relatively large amount of mathematical properties to construct a lattice, and (c) is a relatively simple example to model as it requires integer arithmetic (unlike real arithmetic that we present in the next chapter).

In evaluating and analysing our implementation's performance, we aimed to understand the steps towards improvement tackling the two problems above by comparing the number of solved and unsolved instances with different parameters. When (i) the theory was UFLIA, UFLRA, or EUF (related to (1)), (ii) the lattice's size was 1,21, or 31, and (iii) the feasibility check of counterexamples was enabled or disabled (both (ii) and (iii), related to (2)); the lattice's size was: "1" for methods with summaries but without any lattice, (i.e., user-defined summaries), and "21" or "31" for LB-CEGAR-BMC with a 21- or 31-literal modulo lattice.

We examined via statistical tests the effect of (i), (ii) and (iii) in the next paragraphs, with an initial hypothesis that there was no significant difference in the number of solved instances (our null hypothesis). We first present our statistical test results and later additional graphs to discuss in detail (i), (ii) and (iii). We applied the statistical tests on LB-CEGAR-BMC with different parameters for the comparison of the number of solved and unsolved instances (solved with negative result (SAT), solved with positive result (UNSAT), false results (FALSE-SAT), time-out (TO), out-of-memory (OM), and other errors (Error)) out of 137 benchmarks with the modulo operator[6]. Most datasets in all sets (**A**, **B**, and **C**) did not pass the normality test. Therefore, we applied the Wilcoxon test (a non-parametric t-test). Our goal was to check with which set of parameters (mentioned above) performed best; that is, the method with the largest mean value when there was a significant statistical difference.

Figure 5.9 presents the results of the statistical analyses with Wilcoxon test (fully-paired, two-tailed) as histograms of the mean value with Standard Error Mean (SEM), where statistically significant p-value at $\leq 0.05$, and labels **A**, **B** and **C** refer to the comparison of the effect of the parameters sets in (iii), (ii)

[6]Our datasets contained samples with positive numbers for solved instances and negative numbers for unsolved instances.

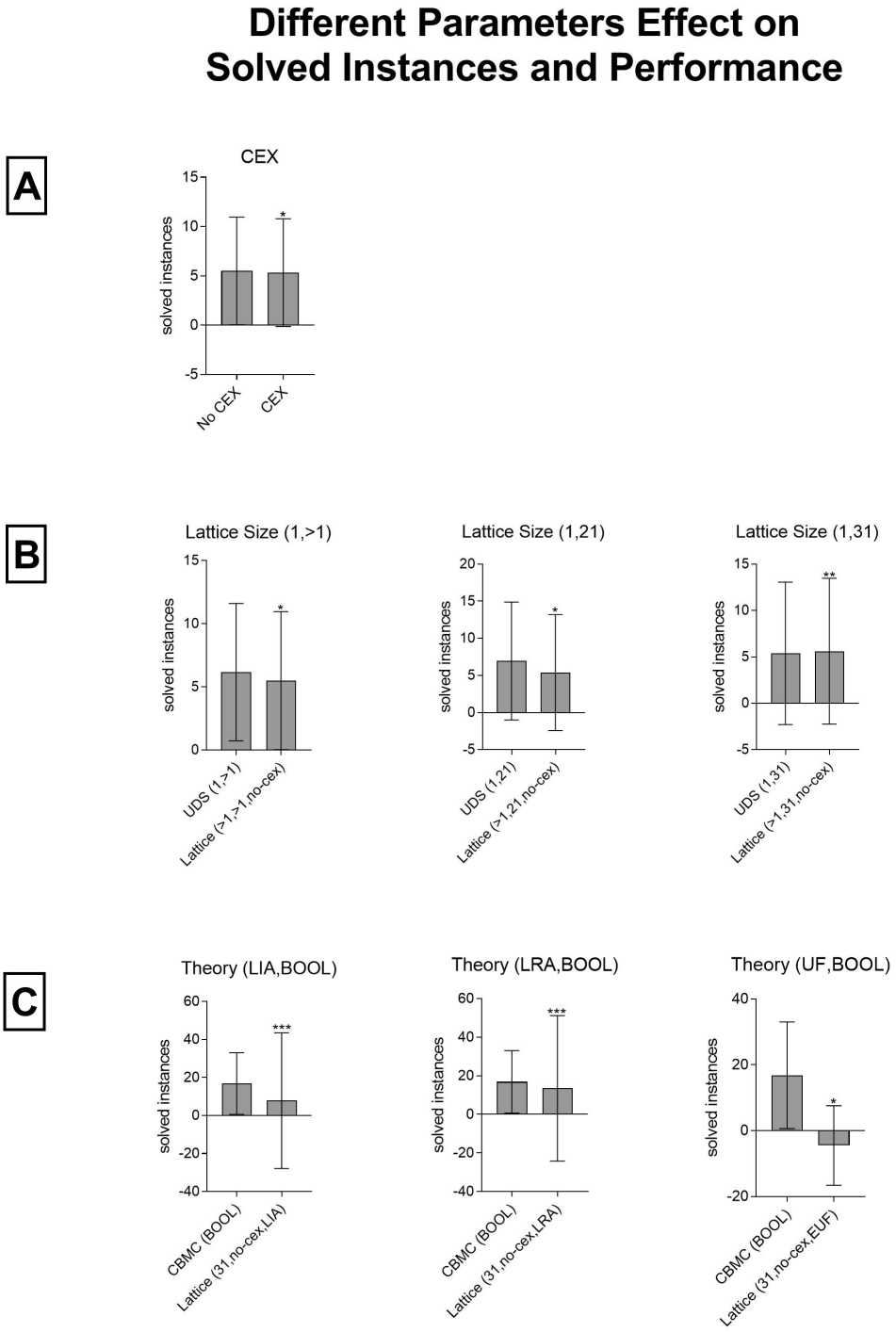


**Figure 5.9:** Graphical description of statistical analyses with Wilcoxon test on the number of solved and unsolved instances (SAT,UNSAT,FALSE-SAT,TO,OM,Error) of LB-CEGAR-BMC with different sets of parameters, presented as histograms of means ± SEM, P-value (top-left): 0.1875, 0.3182, 0.0586, 0.8789, 0.3438, 0.2188, 0.1875, and labels for comparison of the effect of **A** (iii), **B** (ii), and **C** (i). All tests results showed no statistically significant difference (yet, the direction was (*) negative, (**) positive, or (***) unclear).

and (i), respectively. The parameters used to obtain the results are mentioned below each bar: (label **A**) LB-CEGAR-BMC with different lattices and different theories compared with (CEX) or without (NO-CEX) counterexample feasibility checks, (label **B**) LB-CEGAR-BMC and UDS compared with the parameters: lattice size, guarded literals set's size, and additional parameters, and (label **C**) LB-CEGAR-BMC and CBMC compared with different theories (BOOL or other SMT theories) and other parameters for LB-CEGAR-BMC (31 guarded literals and no-cex). The results of LB-CEGAR-BMC are on the right bar for labels **B** and **C**. Label **A** is the comparison of LB-CEGAR-BMC between different parameters; that is, the results of LB-CEGAR-BMC are on both bars.

We observed **no** signification difference in all tests in **A**, **B**, and **C**. While we expected the null hypothesis to hold in **B** and **C**, it was unexpectedly true also for **A** as we predicted the opposite when constructing our method (that is, we did expect counterexamples to make a difference in term of performance). Besides, in **B**, the direction of the results was mixed (positive and negative) with no signification difference. Therefore, we would like to extend the discussion using additional graphs and data in the next paragraphs.

**Evaluation of the counterexamples' guidance on the performance of LB-CEGAR-BMC.** In Fig. 5.10 we compared *(in blue line)* LB-CEGAR-BMC with counterexample feasibility checks (as described in Sec. 5.6.1 with option (5)) against *(in green line)* LB-CEGAR-BMC with no counterexample feasibility check (that is, instead of calling the CEX-validator, we always returned "$false$" and forced refinement via the refiner). The graph in Fig. 5.10 shows no major changes (even slight worse) when used the CEX-validator with respect to the number of solved instances (**SAT** and **UNSAT**), which matches our observation on the

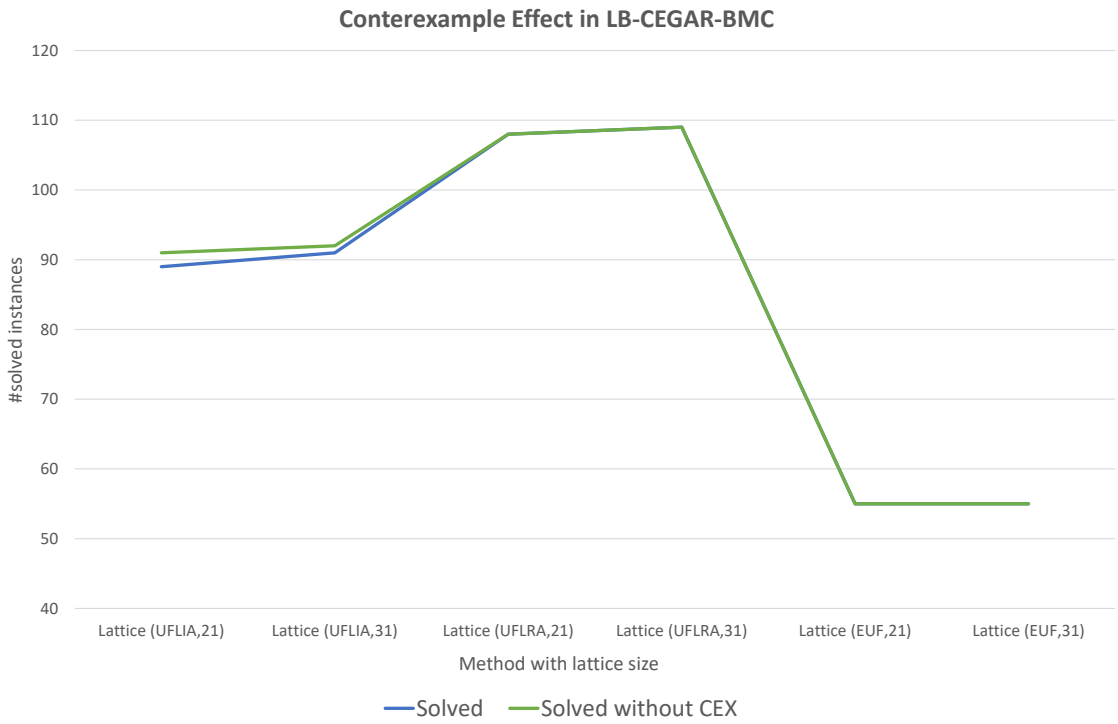


**Figure 5.10:** Effect of counterexample feasibility checks in LB-CEGAR-BMC on performance against LB-CEGAR-BMC without it.

statistical results in Fig.5.9 **A**. The change in the number of solved instances was due to time-out and out-of-memory exceptions when we loaded the same lattices to HiFrog in the evaluation. Hence, we have concluded that the CEX-validator should solve the counterexample feasibility check via a lighter theory than bit-vectors. In Chap. 6, we use NRA for the counterexample feasibility checks; in future work (Chap. 9), we suggest using NRA, NIA or delta-SAT instead.

An interesting observation on Fig. 5.10 is related to the performance of the benchmarks in UFLIA versus UFLRA. The LB-CEGAR-BMC performed better with UFLRA due to out-of-resources exceptions in the instances solved with UFLIA (as shown in Fig. 5.11, right graph, blue line). This is generally expected behaviour, as SMT solvers tend to implement the integer arithmetic solver on

top of the real arithmetic solver, which means that the former consumes more resources than the latter.

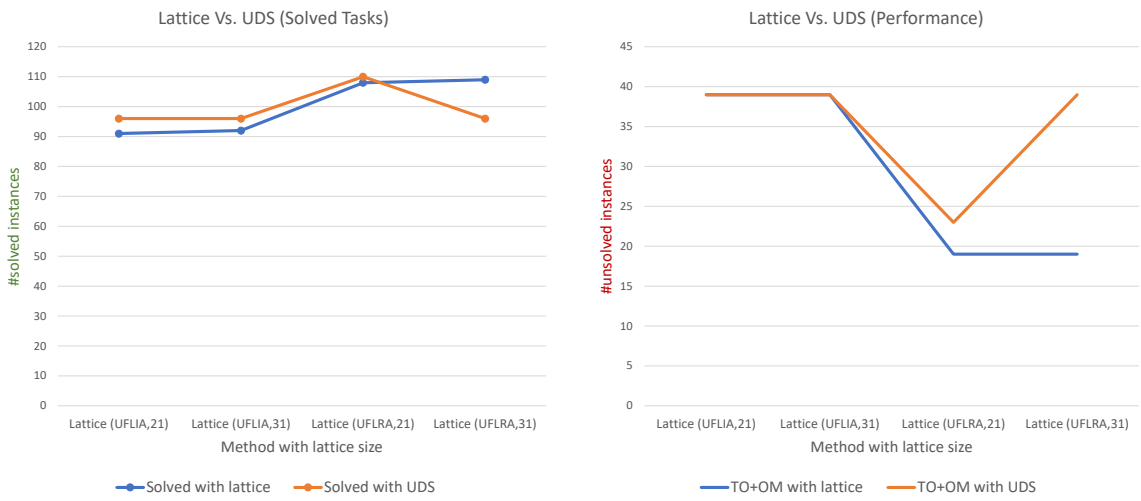


**Figure 5.11:** Measure the effect of lattice traversal algorithm in LB-CEGAR-BMC on performance against flat lattice.

**Evaluation of LB-CEGAR-BMC against UDS in HiFrog.** In Fig. 5.11, we compared the performance of LB-CEGAR-BMC against HiFrog with user-defined summaries (UDS). The set contained either 21 or 31 summaries; in UDS, we loaded all summaries at once to HiFrog. Fig. 5.11 (left) shows that LB-CEGAR-BMC (in blue line) and UDS (in orange line) solved on average the same number of instances (**SAT** and **UNSAT**), which supports our observation in Fig.5.9 **B**, that there was no signification difference between the methods with modulo operator. We have observed the following from Fig. 5.11:

- LB-CEGAR-BMC (in blue line) had fewer instances with out-of-resources exceptions than UDS (in orange line) on the average (Fig. 5.11, right graph).
- The resources consumption problem (that is, out-of-memory and out-of-

time) has grown in UDS with the size of the set of summaries loaded to HiFrog. We did not observe this behaviour in LB-CEGAR-BMC[7].

- LB-CEGAR-BMC with lattices of 21 and 31 guarded literals had the same resource consumption in a specific theory.

Following the above, we observed that the memory and time consumption in LB-CEGAR-BMC was better and more efficient on average than in HiFrog with UDS. Moreover, the use of the structure of a lattice assisted keeping the resources consumption low even when the size of the lattice grew (with UDS, the tool's performance has gotten worse with the increase in the number of summaries loaded to HiFrog). To conclude, unlike UDS, due to the use of the structure of a lattice, our method could handle a larger amount of summaries without compromising performance, even-though HiFrog with UDS is a well-tested functionality taken from FunFrog, while our method is a new prototype with much more complex functionality than UDS.

**Evaluation of LB-CEGAR-BMC with different theories.** Fig.5.9 **C** showed no significant statistical difference, usually with an unclear direction of the results between LB-CEGAR-BMC and CBMC when solving the benchmarks with modulo function. We referred to each row in our datasets in this paragraph to analyse and understand better the results of solved and unsolved instances, separately. In Fig. 5.12 we compared the result of the evaluation of LB-CEGAR-BMC with different theories (UFLIA-Lattice-Ref. label, navy-blue bars, UFLRA-Lattice-Ref. label, blue bars, and EUF-Lattice-Ref. label, orange bars, all with lattice of 31 guarded literals for the modulo operation) on solved-safe instances

[7]This method is not suitable for a one-time verification task. We suggested here to apply this method only for common functions required for many verification tasks. Hence, we did not consider the lattice construction time.

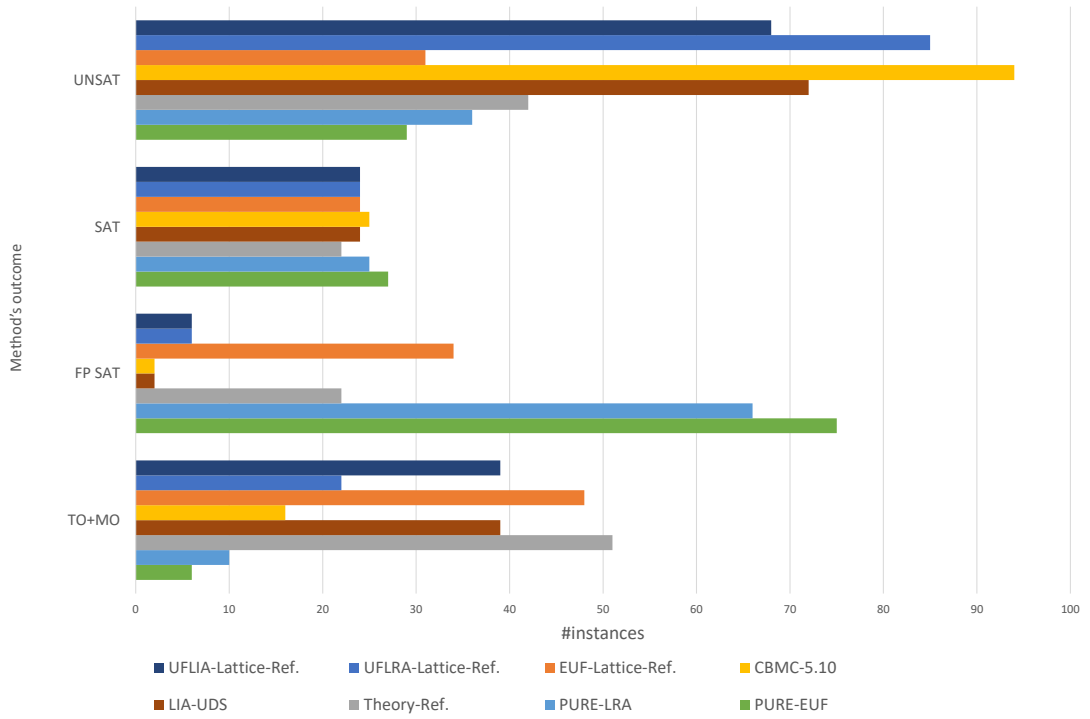


**Figure 5.12:** Experimental results of the LB-CEGAR-BMC approach against HiFrog (various parameters) and CBMC.

(UNSAT bars), solved-unsafe instances (SAT bars), unsolved-unsafe instances[8] (FP SAT bars), and unsolved instances due to out of resource errors (TO+MO bars). The comparison was against other approaches: CBMC with the default settings (CBMC-5.10 label, yellow bars), HiFrog with 31 user-defined summaries (LIA-UDS label, brown bars), Theory refinement (Theory-Ref. label, grey bars), HiFrog with LRA (PURE-LRA label, light-blue bars), and HiFrog with EUF (PURE-EUF label, green bars). Experiments with LIA was not included in the comparison as this option in HiFrog is still experimental.

**(UNSAT bars)** LB-CEGAR-BMC with UFLRA performed quite well; it solved 85 safe instances when CBMC solved 94 (the highest number). LB-CEGAR-BMC with UFLIA performed almost as good as HiFrog with user-

[8]That is, instances where the expected result was **UNSAT** but an approach returned **SAT** and hence the tool reported safe benchmarks as unsafe.

defined summaries. However, both of them (UFLIA-Lattice-Ref. and LIA-UDS) solved less safe instances in comparison to the performance of these settings with linear real arithmetic. This was due to performance issues of the linear integer arithmetic we observed before (see the discussion for Fig. 5.10 and Fig. 5.11).

**(SAT bars)** All approaches (we tried in this section) solved similar instances in the unsafe set, where HiFrog with EUF out-performed all the other approaches as expected. EUF is a light-theory and was the lightest in comparison to the rest of the theories and techniques we used. Hence we expected it to have the highest number of solved-unsafe instances. On the other hand, HiFrog with EUF also had the highest number of false results.

**(FP SAT bars)** The unsolved instances with false results were mainly affected by the availability of the definition of the modulo operation (full or partial) and the ability to model arithmetic operations in general (e.g., to model $<$ in EUF or $a{*}b$ in LRA). Consequently, LB-CEGAR-BMC with EUF, HiFrog with LRA and HiFrog with EUF had the highest number of false results; all these approaches ran either with no additional user-defined libraries or with theories that have no support for arithmetic operations.

**(TO+MO bars)** LB-CEGAR-BMC ran out-of-resources as HiFrog with UDS (with 31 summaries and linear integer arithmetic). The theory refinement approach (with similar counterexample feasibility check) had the highest number of out-of-resources instances, while HiFrog had the lowest number of out-of-resources instances.

We can already conclude the following from the statistical tests and the three paragraphs discussing the results compared LB-CEGAR-BMC with different parameters (i), (ii) and (iii):

- A further investigation is required for an efficient counterexample feasibility check.
- Loading summaries into HiFrog affected the performance in a negative way in compare to approaches with no summaries.
- A definition (even if partial) of arithmetic operation and modulo operation was essential for solving the benchmarks here, which was the reason CBMC out-perform (in several instances) our current implementation of LB-CEGAR-BMC. However, we demonstrated the potential of using a lattice and light-weight theories over the full definition of a library function for the majority of the instances CBMC solved.

We present below the full evaluation in Table 5.2.

**Full results of the evaluation of LB-CEGAR-BMC.** In Table 5.2, the column *Approach* refers to the approach we used to get the results in a row, and the column *logic* indicates which SMT logic used. The column *#lit* states the number of guarded literals or summaries loaded into the tool or N/A if no summaries used, and the column *Opts.* indicates if there is any specific tool's option we used (NO-CEX for disabling the counterexample feasibility checks, UDS for loading user-defined summaries into the tool, and Theory Ref. to use the theory refinement option in HiFrog). For the results, the column *#instances solved* gives the number of correctly solved-unsafe and solved-safe instances, and last, the column *#instances unsolved* gives the number of unsolved instances for different reasons (that is, instances with false positive, time-out, out-of-memory, and other errors results). We present the full evaluation of LB-CEGAR-BMC with the following settings: (i) with LIA with EUF (UFLIA) encoding with 21 or 31 guarded literals in the lattice for modulo operation (rows 1-2) and without counterexample

**Table 5.2:** Full verification results of LB-CEGAR-BMC against CBMC, theory refinement, UDS, and EUF and LRA without lattices (#: number of instances, FP SAT: false positive SAT result, TO: time out of 4000s, MO: Out of Memory of 3GB, and ERR: other exceptions and errors).

| **Approach** | **logic** | **#lit** | **Opts.** | **#instances solved** <SAT,UNSAT> | **#instances unsolved** <FP SAT,TO,OM,ERR> |
|---|---|---|---|---|---|
| LB-CEGAR-BMC | LIA+UF | 21 | N/A | <23,66> | <8,39,0,1> |
| LB-CEGAR-BMC | LIA+UF | 31 | N/A | <23,68> | <6,39,0,1> |
| LB-CEGAR-BMC | LRA+UF | 21 | N/A | <24,84> | <7,0,21,1> |
| LB-CEGAR-BMC | LRA+UF | 31 | N/A | <24,85> | <6,0,19,3> |
| LB-CEGAR-BMC | EUF | 21 | N/A | <24,31> | <34,6,42,0> |
| LB-CEGAR-BMC | EUF | 31 | N/A | <24,31> | <34,6,42,0> |
| LB-CEGAR-BMC | LIA+UF | 21 | NO-CEX | <24,67> | <7,39,0,0> |
| LB-CEGAR-BMC | LIA+UF | 31 | NO-CEX | <24,68> | <6,39,0,0> |
| LB-CEGAR-BMC | LRA+UF | 21 | NO-CEX | <24,84> | <7,0,19,3> |
| LB-CEGAR-BMC | LRA+UF | 31 | NO-CEX | <24,85> | <6,0,19,3> |
| LB-CEGAR-BMC | EUF | 21 | NO-CEX | <24,31> | <34,6,42,0> |
| LB-CEGAR-BMC | EUF | 31 | NO-CEX | <24,31> | <34,6,42,0> |
| HiFrog | LIA | 21 | UDS | <24,72> | <**2,**39,0,0> |
| HiFrog | LIA | 31 | UDS | <24,72> | <**2,**39,0,0> |
| HiFrog | LRA | 21 | UDS | <24,86> | <**2,**6,17,2> |
| HiFrog | LRA | 31 | UDS | <24,72> | <**2,**10,29,0> |
| HiFrog | EUF | 21 | UDS | <24,38> | <27,6,42,0> |
| HiFrog | EUF | 31 | UDS | <24,38> | <27,6,42,0> |
| HiFrog | CUF | 0 | Theory Ref. | <22,42> | <22,10,1,40> |
| HiFrog | LRA | 0 | N/A | <25,36> | <66,10,0,0> |
| HiFrog | EUF | 0 | N/A | <27,29> | <75**,6,0,0**> |
| CBMC 5.10 | | 0 | N/A | <**25,94**> | <**2,**6,9,1> |

feasibility check (rows 7-8), (ii) with LRA with EUF (UFLRA) encoding with 21 or 31 guarded literals in the lattice for modulo operation (rows 3-4) and without counterexample feasibility check (rows 9-10), and (iii) with EUF encoding with 21 or 31 guarded literals in the lattice for modulo operation (rows 5-6) and without counterexample feasibility check (rows 11-12). We compared our implementation of LB-CEGAR-BMC approach in HiFrog (in rows 1-12) against:

1. HiFrog with LRA encoding and EUF encoding (Chap. 4). In rows: HiFrog LRA 0 N/A (row 20) and HiFrog EUF 0 N/A (row 21) with the OpenSMT2 SMT solver.

2. HiFrog with user-defined summaries with LRA, LIA, or EUF encoding (Chap. 4), with either 21 or 31 summaries (in rows: HiFrog LIA 21 UDS (row 13), HiFrog LIA 31 UDS (row 14), HiFrog LRA 21 UDS (row 15), HiFrog LRA 31 UDS (row 16), HiFrog EUF 21 UDS (row 17), and HiFrog EUF 31 UDS (row 18)), and with the OpenSMT2 SMT solver for the counterexample feasibility checks and for verification with EUF, and the Z3 SMT solver for verification with LRA and LIA.

3. HiFrog theory-refinement option with CUF encoding (Chap. 7). In row: HiFrog CUF 0 Theory Ref. (row 19) with the OpenSMT2 SMT solver.

4. CBMC version 5.10 [CKL04] (the winner of the software model checking competition falsification track in 2017). In row: CBMC 5.10 0 N/A (row 22).

   Note that, CBMC version 5.10 `-refine` option performs as the standard CBMC version, and thus is not included in Table 5.2.

Even with a prototype implementation of our algorithms, LB-CEGAR-BMC

performed quite well in comparison to established tools, with 109 solved instances (for UFLRA, 31 guarded literals) and 91 solved instances (for UFLIA, 31 guarded literals) versus 119 solved instances with CBMC, 61 solved instances with HiFrog LRA, and 56 solved instances with HiFrog EUF. LB-CEGAR-BMC with EUF performed poorly even with respect to HiFrog EUF. In general, the LB-CEGAR-BMC approach (with any of the theories) failed to prove safety once other operations abstracted from the SMT encoding (e.g., $SHL$, $SHR$, pointer arithmetic) or, in UFLRA and UFLIA when the code contained non-linear expressions.

The experimental results of LB-CEGAR-BMC (with and without counterexample feasibility checks) and UDS mode of HiFrog were pretty much the same, with 109 solved instances (for UFLRA, 31 guarded literals), and 91 solved instances (for UFLIA, 31 guarded literals), versus HiFrog with UDS with 96 solved instances (for LRA and LIA 31 summaries). However, in overall, it had better resource consumption than UDS mode of HiFrog (TO+OM instances). An interesting result was the performance of HiFrog with UDS in LRA with 21 and 31 summaries, with the former out-performed the latter due to out-of-memory exception (occurred in many instances) when loading 31 summaries to the tool. LB-CEGAR-BMC has no instance with time-out exception with UFLRA, while all the rest of the approaches had (even for HiFrog with a lightweight theory only as PURE LRA and PURE EUF).

LB-CEGAR-BMC, out-performed theory refinement mode of HiFrog; when both theory refinement and LB-CEGAR-BMC (rows 1-6) used the same counterexample feasibility check. LB-CEGAR-BMC with no counterexample feasibility check (rows 7-12) also out-performed theory refinement mode of HiFrog. Theory refinement had a high rate of instances with a general exception (unneces-

sarily time-out or out-of-memory), which might indicate that the implementation of theory refinement was not mature enough.

Last observation is related to the performance of LB-CEGAR-BMC with UFLRA versus UFLIA, with the former out-performed the latter; we expected the opposite results since these benchmarks contained mainly integer problems.

To conclude from the above, LB-CEGAR-BMC had similar results to CBMC and HiFrog (with different settings) but failed to prove safety when other operations abstracted from the SMT encoding. A possible solution to this problem can be using lattices to refine these operations in addition to modulo operation. We reported a better resource consumption in LB-CEGAR-BMC with a copy of a lattice per occurrence of the modulo function than with user-defined summaries, especially as the number of summaries loaded to HiFrog had grown. Last, the counterexample feasibility checks were inefficient as there was no major difference in the results with this option or without it. Even with the improvement of the implementation of the counterexample feasibility checks, we believe these checks at the bit level might still be expensive, and suggest to handle this problem by using either a different theory or a solving approach for these checks.

None of the approaches in the comparison reports unsafe benchmarks as safe. The full table of results and the set of benchmarks are available at [HiF19].

**Experiment settings.** We used two different in size meet semi-lattices for the refinement of programs in C with modulo function calls. We constructed these lattices with $F_{mod}$ a set of 21 or 31 guarded literals, which were small arbitrary subsets of the modulo function properties. The width and height of these lattices were 10 and 13, and 11 and 13, respectively. The raw data was taken from the

Coq proof assistant [The19a] and Wikipedia [Mod19] (see [HiF19] for a meet semi-lattice sketch).

We used the OpenSMT2 SMT solver [HMAS16] for EUF solving and counterexample feasibility checks, and the Z3 SMT solver [DMB08b] for LIA with EUF and LRA with EUF solving. Note that, we removed **three** guarded literals from the original set of guarded literals $F_{mod}$ to prevent contradictions with IEEE 754 definition (done for any guarded literal that was based on the pure algebraic definition or the Euclidean definition [Bou92]).

Our benchmarks consisted of 137 C programs with the modulo operation (possibly with more than one assertion). The benchmarks set was a mix of 81 SV-COMP [Com18] benchmarks and 56 of our benchmarks (either from HiFrog tool, previous papers or crafted benchmarks with modulo operation for this work). Majority of the benchmarks included at least several statements in the code with the modulo operation and loops.

# Chapter 6

# Lattice-based SMT for Program Verification

We present a lattice-based satisfiability modulo theory for verification of programs with library functions, for which the mathematical libraries supporting these functions contain a high number of equations and inequalities. Common strategies for dealing with library functions include treating them as uninterpreted functions or using the theories under which the functions are fully defined. The full definition could in most cases lead to instances that are too large to solve efficiently.

Our lightweight theory uses lattices for efficient representation of library functions by a subset of guarded literals. These lattices are constructed from equations and inequalities of properties of the library functions. These subsets are found during the lattice traversal. We generalise the method to a number of lattices for functions whose values depend on each other in the program, and we describe a simultaneous traversal algorithm of several lattices, so that a combination of guarded literals from all lattices does not lead to contradictory values of their variables.

We evaluate our approach on benchmarks taken from the robotics community, and our experimental results demonstrate that we are able to solve a number of instances that were previously unsolvable by existing SMT solvers.

## 6.1 Introduction

Finding a scalable way for verifying programs or systems which use library functions as a main part of their application (e.g., implementation of robots' movements in the Robot Operating System (ROS) [ROS19]) is a non-trivial task: the code may contain hundreds of interacting expressions of the properties of the library functions, whose truth values depend on each other. A straightforward solution would be to use increasingly precise theories. However, this approach results in prohibitively expensive computations (e.g., by adding details at the bit-level to describe trigonometric functions, which would be very expensive).

Trigonometric functions serve as a good illustration of the problem outlined above, as many domains of application, such as robotics, planning [Wit16], and simulations for physics and engineering [osp19], rely on the computation of trigonometric functions. Verification of software using trigonometric library functions [AP10, DMP13, DM14, TE14, CGI+17b, GJBF18] either requires a large amount of numerical calculations of polynomials along with irrational numbers or uses large look-up trigonometric tables which tend to be less precise and are memory consuming [KGP06]. The former technique usually replaces the irrational expressions with rational expressions with a defined error bound [MM05, DLM09, AP10, CGI+17b, CGI+18b] in order to bound or to evaluate trigonometric expressions to some precision. A more precise approach relies on Taylor series representation of trigonometric functions over reals; as it leads

to complex computations, the resulting instances are too large to solve efficiently for all, but very small programs.

Finally, the solver implemented in HiFrog (Chap. 4, [AAC+17]) supports the addition of sets of equations and inequalities as *user-defined function summaries*. We can, therefore, extract the known properties of library functions from the external libraries and encode them as user-defined SMT summaries to pass to HiFrog, and, ultimately, to the SMT solver. However, this approach is not scalable either, as we do not know beforehand which properties are going to be relevant for solving a particular instance. Hence, for library functions with a large number of equations (such as trigonometric functions—there are many equations describing properties of these functions on some subdomains), the user-defined summaries will render the instance too large to be solvable efficiently (or at all).

In this chapter, we present a novel approach to reasoning about programs whose correctness depends on the values of library functions. Our approach uses the concept of *subset lattices* to construct an efficient representation of known properties of these functions. Essentially, we order the set of subsets of equations describing properties of library functions in a lattice, where each element corresponds to a set of properties that hold for some subdomain of the inputs. At every iteration of the algorithm, we verify the program with only a subset of equations that corresponds to the current element in the lattice. If this subset is insufficient for the verification (that is, does not provide enough information about the library function), we *refine* it by traversing the lattice to a higher element, containing a superset of the equations.

This lattice-based counterexample-guided abstraction refinement algorithm (LB-CEGAR) is based on the traditional counterexample-guided abstraction refinement (CEGAR) [CGJ+00, CGJ+03], but replaces the refinement of the theory

by the refinement of the set of equations for the library function in the program. Our approach is similar to the traditional CEGAR approach in the sense that a SAT result may indicate a real counterexample (in which case there are concrete values of symbolic variables that show the existence of this execution), or a *spurious* counterexample, where the satisfying assignment provided by the solver is due to over-approximation in the representation of the program. In contrary to the traditional CEGAR, where an UNSAT result indicated that there are no counterexamples in an abstraction of the program and hence in the concrete program as well, in LB-CEGAR the UNSAT result merely means that there are no counterexamples in the current subdomain of the input to the library function. As we describe in the chapter, the lattice is constructed so that every *lattice frontier* covers the whole domain of the input variables. Hence, in case of an UNSAT result, the LB-CEGAR algorithm attempts to construct a *frontier of unsatisfiability*. Such a frontier would indicate that there are no counterexamples in the current abstraction for each subdomain of the input, and hence for the whole domain as well.

In the previous chapter, we described a simplified LB-CEGAR algorithm for the case of one library function in the program and for small lattices. In this chapter, we extend LB-CEGAR to the general case, where the program may contain several library functions whose values can be interconnected (for example, $\sin x$ and $\cos x$). Furthermore, each function can appear in the program multiple times, thus inducing several instances of the lattice, which are traversed simultaneously. We describe the generalised LB-CEGAR algorithm and analyse its worst-case complexity and heuristics in Sec. 6.3.

We implemented the generalised LB-CEGAR algorithm in the bounded model checker HiFrog (Chap. 4, [AAC+17]) supporting a subset of the C language and

using the OpenSMT2 SMT solver [HMAS16] and the Z3 SMT solver [DMB08b] and evaluated the implementation on a large set of benchmarks containing programs whose correctness depends on the values of trigonometric functions. The experimental results clearly demonstrate an advantage to LB-CEGAR over other approaches. We outline the implementation in Sec. 6.4.1 and the experimental results in Sec. 6.4.2.

Our results are based on the trigonometric functions being treated as uninterpreted functions in the encoding of the problem to the SMT solver and the encoding of the mathematical equations as user-defined function summaries in the semantics of reals, which is common in modelling software [AMP06, KRSS16, AAC+17, BG18]. We assume the correctness of these equations (such as $\sin^2 x + \cos^2 x = 1$) over real numbers. The challenge of verifying problems over IEEE floating-point semantics, stemming from the implementation of the trigonometric functions in the underlying architecture, is outside of the scope of this thesis. There are clear advantages to pinpointing the subset of mathematical equations that are instrumental for the correctness of the program under verification (which is what we do in this chapter) to the challenge of verification over floating-point semantics, and we leave the exploration of this direction for future work (see Chap. 9, Sec. 9.2, last paragraph).

The following example illustrates the motivation for LB-CEGAR on a small program with trigonometric functions.

**Example 8.** The program in Fig. 6.1 contains two library function calls: $\sin x$ and $\cos x$. The correctness of the program follows immediately from the following trigonometric identity:

$$\forall x \in \mathbb{R}. \sin^2 x + \cos^2 x = 1. \tag{6.1}$$

```
1  #include <math.h>
2
3  double nonlin(double x) {
4      double x_sin = sin(x);
5      double x_cos = cos(x);
6      return x_sin*x_sin + x_cos*x_cos;
7  }
8
9  void main() {
10     double y = nondet();
11     double z = nonlin(y);
12     assert(z == 1);
13 }
14
```

**Figure 6.1:** Program with two different library functions.

Clearly, verifying a program with $\sin x$ and $\cos x$ treated as uninterpreted functions (that is, having non-deterministic values) would result in numerous spurious counterexamples. LB-CEGAR overcomes this problem by representing some salient properties of these functions as lattices of equations, including, in particular, Eq. (6.1).

In this case, the elements of the lattices for $\sin x$ and for $\cos x$ at each iteration of LB-CEGAR are not independent, as Eq. (6.1) should hold for each combination of these elements. Moreover, having a lattice only for one function would not suffice for proving the correctness of this program, as then we would have Eq. 6.1 only for one of these functions, while leaving the other one as a non-deterministic variable. This would lead to spurious counterexamples, stemming from assigning illegal values to the non-deterministic variable (for example, if $\cos x$ is left as a non-deterministic variable, it could be assigned the value 2, hence falsifying the assertion).

General examples of lattice construction with different properties of library functions as well as the refinement with such lattices of a small code example in C are available at [HiF19].

The implementation, the set of benchmarks, the experimental results, and additional materials, are available at [Git19a, Git19b, HiF19].

## 6.2 Preliminaries

**Lattices of Guarded Literals** We use the definitions in Sec. 5.3.1 and construct the lattice as described in Sec. 5.4. We do not intend to extend the discussion of the lattice construction in this chapter. See Sec. 2.1 for general definitions of a poset, a lattice and a semi-lattice.

**Function summaries.** We exploit this functionality by providing HiFrog with the library of *user-defined* summaries (Sec. 2.5.1) derived from external libraries for the functions, whose values are critical for determining correctness of the program, and we organise them in lattices as we explain in Chap. 5. This allows us to verify programs in the most abstract theory of equality logic with uninterpreted functions (EUF) or its extensions.

It is important to note that the use of function summaries is a choice of implementation. There is no limitation of our approach using function summaries as a means of input over other formats (e.g., plain strings).

## 6.3 The Lattice-based Counterexample-Guided Abstraction Refinement (LB-CEGAR) Algorithm

In this section we present the main contribution of the chapter—the LB-CEGAR algorithm. We start with an informal overview and then present the formal description of the algorithm. We proceed with discussing its worst-case complexity and then present several heuristics that reduce the complexity for the majority of the cases.

### 6.3.1 Overview of the LB-CEGAR algorithm

The inputs to the Lattice-based Counterexample-Guided Abstraction Refinement (LB-CEGAR) algorithm (Alg. 5) are a bounded loop-free program $P$ that includes the function $f$ and a safety property $t$. The algorithm follows the standard procedure of translating $P$ and the negation of $t$ to a first-order formula $\varphi$ and invoking an SMT solver in order to find a satisfying assignment. In contrast to the standard approach, in LB-CEGAR the SMT solver has access, in addition to $\varphi$, to the external lattice $L_f$ of guarded literals for $f$ constructed in Sec. 5.3.1. At each iteration LB-CEGAR adds the set of guarded literals in the current element $E$ of this lattice to $\varphi$ before sending $\varphi$ to the SMT solver.

The refinement loop in LB-CEGAR, invoked when a satisfying assignment does not correspond to a concrete counterexample, amounts to the traversal of $L_f$ as described below in the procedure $traverse_{SAT}$.

The algorithm terminates when it either finds a satisfying assignment that corresponds to a concrete counterexample (and hence a bug in $P$), reaches all

maximal elements of $L_f$ without finding concrete counterexamples for any of the satisfying assignments (that is, the current set of properties of $f$ encoded in $L_f$ is insufficient to verify $P$), or finds a *frontier* of $L_f$ such that $\varphi$ is unsatisfiable with each element of the frontier separately. The latter case implies that there are no counterexamples in the over-approximation of $P$ for the whole domain of the inputs, and hence $P$ satisfies $t$.

An iteration of LB-CEGAR with a program $P$, a safety property $t$, and a current element $e$ consisting of the set of guarded literals $S(e)$ of the lattice $L_f$ results in one of the following (for one library function $f$ and a single occurrence of $f$ in the loop-free program $P$):

- An SMT solver finds a satisfying assignment for $\varphi$ with $S(e)$, and there is a concrete counterexample corresponding to this assignment. The algorithm terminates, outputting the counterexample as an evidence of the negative result of model checking $P$.

- An SMT solver finds a satisfying assignment for $\varphi$ with $S(e)$, but there is no concrete counterexample corresponding to this assignment. The algorithm invokes a *refinement step* that amounts to traversing $L_f$ to an element $e'$ that refines $e$, that is, $S(e) \subset S(e')$. If no such element exists (in other words, $e$ is a maximal element of $L_f$), the algorithm terminates with inconclusive results.

- An SMT solver returns the UNSAT result for $\varphi$ with $S(e)$. In other words, there is no satisfying assignment to $\varphi$ in the subdomain of inputs induced by $e$. The refinement step of LB-CEGAR is, then, to check satisfiability of $\varphi$ with elements of $L_f$ that complement the subdomain of $e$ to the whole

domain of the input (that is, with elements of $L_f$ that together with $e$ form a *frontier* of $L_f$).

- An SMT solver returns the UNSAT result for $\varphi$ with $S(e)$, and $e$ is a part of a frontier of $L_f$ for which $\varphi$ is unsatisfiable. This result implies that there is no satisfying assignment to $\varphi$ over the whole domain of the inputs, and therefore $P$ is safe with respect to $t$.

If the function $f$ appears in $P$ several times, an instance of $L_f$ is created for each occurrence. Furthermore, if $P$ contains more than one library function for which we have a lattice of guarded literals, all these lattices are incorporated in LB-CEGAR. For programs with trigonometric functions, which are the primary domain of application in this chapter, it is often the case that an equation includes several functions—see, for example, the program in Ex. 8.

In the next section, we present a pseudo-code for LB-CEGAR and discuss the general case of several functions and several occurrences of each function in the program.

### 6.3.2 The main LB-CEGAR algorithm

The pseudo-code of LB-CEGAR is presented below. The input to the algorithm is a loop-free program $P$, a safety property $t$, and a set of lattices *Lattices*.

The sub-procedures and notations in Alg. 5 are defined as follows.

- The sub-procedure $checkSAT(\psi)$ determines the satisfiability of an input formula $\psi$ in a given logic via an SMT solver.

- The sub-procedure $checkRealCE(P, t, CEX)$ returns **true** if *CEX* can be

**Algorithm 5:** LB-CEGAR

**Input :** Program $P$, safety property $t$, and set *Lattices*
**Output:** $\langle$**Safe**$\rangle$, $\langle$**Unsafe**, *CEX*$\rangle$, or $\langle$**Unknown**, $\perp\rangle$

$\varphi \leftarrow P \wedge \neg t$
*Query* $\leftarrow \varphi$
$\langle result, CEX\rangle \leftarrow checkSAT(Query)$
**if** *result is* **UNSAT** $\vee$ $checkRealCE(\varphi, CEX)$ **then**
  **go to** `Exit` // No lattice-based refinement needed
**end**
$\chi \leftarrow$ **true**
**repeat**
  $\chi' \leftarrow \chi$ // Formula from the previous iteration
  **if** *result is* **UNSAT then**
    $traverse_{UNSAT}(Lattices)$
  **end**
  **if** *result is* **SAT then**
    $traverse_{SAT}(Lattices)$
  **end**
  $\chi \leftarrow currentFacts(\varphi, Lattices)$
  // Solve again if there are new literals
  **if** $\chi \neq \chi'$ **then**
    *Query* $\leftarrow \varphi \wedge \chi$
    $\langle result, CEX\rangle \leftarrow checkSAT(Query)$
  **end**
**until** $(\chi == \chi') \vee \vee checkRealCE(Query, CEX) \vee termination(result, Lattices)$;
`Exit:` // End of LB-CEGAR
**if** *result is* **UNSAT then**
  **return** $\langle$**Safe**$\rangle$ // Safe
**end**
**if** $checkRealCE(P, t, CEX)$ **then**
  **return** $\langle$**Unsafe**, *CEX*$\rangle$ // Real counterexample
**end**
**return** $\langle$**Unknown**$\rangle$ // Inconclusive results, further refinement needed

concretised to a counterexample[1], demonstrating a behaviour of $P$ that falsifies $t$.

- The set *Lattices* consists of all occurrences of lattices for all library functions in $P$.
- We define $currentFacts(\varphi, Lattices)$ as $\varphi \wedge currentFacts(Lattices)$.
- We denote by $L_f^i$ a lattice for the $i$th occurrence of $f$ in $P$, and by $e$ the

[1]A counterexample (conjoined with the model) is tested by using a theory under which the library function is fully defined.

current element in the lattice traversal. (i) For an element $e$, we define *currentFacts*(*Lattices*) as the *conjunction of guarded literals* of $e$. (ii) For a lattice with a frontier, we define *currentFacts*(*Lattices*) as the disjunction of the conjunction of guarded literals of elements in the frontier[2].

- The sub-procedure $traverse_{UNSAT}(Lattices)$ performs the traversal of the lattice from the current element $e$ to the next element $e'$ if the result of model checking $\varphi \wedge currentFacts$ is **UNSAT**. The next element $e'$ in the same lattice as $e$ is $e$'s 'sibling', that is, an element, whose set of literals corresponds to a different subdomain of the input. If there is already a frontier of elements in each lattice such that model checking $\varphi \wedge currentFacts$ returns **UNSAT** for each element of these frontiers, the procedure $traverse_{UNSAT}(Lattices)$ does not change the current element $e$.

- The sub-procedure $traverse_{SAT}(Lattices)$ is invoked when there is a satisfying assignment for $\varphi \wedge currentFacts$, but the counterexample induced by it is *spurious*, that is, it does not correspond to a behaviour of $P$ falsifying $t$. The procedure traverses the lattice to an element $e'$ that refines $e$, that is, $S(e) \subset S(e')$. If $e$ is a maximal element, the procedure $traverse_{SAT}(Lattices)$ does not change the current element $e$.

- In both sub-procedures $traverse_{UNSAT}$ and $traverse_{SAT}$, the lattices are traversed either in an arbitrary order or in an order determined by heuristics. However, we do not describe a fix order mechanism of traversing the lattice to be able to deal with different combinations of library functions. We describe such heuristics in Sec. 6.4.1.

[2]The formula of the frontier is a first-order formula in DNF form. For example, given a frontier $X(L_f^i)$, its first-order formula is: $\vee_{e \in X(L_f^i)} . \wedge_{l \in e} l$.

- The sub-procedure *termination*(*result*, *Lattices*) checks whether one of the termination conditions holds: either the current satisfying assignment induces a concrete counterexamples, or there is an **UNSAT** frontier for each lattice $L_f^i \in$ *Lattices*, or there is a satisfying assignment for each maximal element in each lattice in *Lattices* that does not induce a concrete counterexample.

Finally, we address the complexity resulting from having several functions in $P$, whose lattices refer to each other. This is illustrated by Ex. 8, where the correctness of the program depends on the guarded literal

$$(assume(\mathbf{true})) \wedge (\sin^2 x + \cos^2 x) = 1.$$

In fact, this is quite common in programs with trigonometric functions, as trigonometric identities often refer to several functions in the same identity. The algorithm identifies library functions used in the set *Lattices* and assigns the same variable to all occurrences of the same function, hence connecting between the lattices of different functions.

### 6.3.3 Correctness and complexity

It is easy to see that LB-CEGAR terminates (Lemma 3 in Sec. 6.3.3). Indeed, the lattice traversal visits every combination of elements of lattices in *Lattices* at most once, and for each combination of elements it invokes the model checking procedure of a bounded loop-free program $P$ with respect to $t$, which terminates. The number of possible combinations of elements in the lattices is exponential in the number of lattices, hence leading to the complexity result below.

**Theorem 1.** *The worst-case running-time complexity of LB-CEGAR is* $O(|L|^n \times MC(P,t))$*, where* $|L|$ *is the bound on the size of each lattice in the set Lattices,* $n$ *is the number of lattices in Lattices, and* $MC(P,t)$ *is the running-time complexity of model checking* $P$ *with respect to* $t$ *using the guarded literals.*

*Proof.* From Lemma 3, we know that the LB-CEGAR terminates. Before termination, the lattice traversal visits every combination of elements of lattices in *Lattices* at most once; since, for the general case, we assume no heuristic or optimization are used during the traversal.

For each combination of elements, it invokes the model checking procedure of a bounded loop-free program $P$ with respect to $t$, which terminates. The number of possible combinations of elements in the lattices is exponential in the number of lattices, hence leading to the complexity result below. □

Moreover, the following theorem states that LB-CEGAR produces a correct result.

**Theorem 2.** *The following holds for any bounded loop-free program* $P$ *and a safety property* $t$*, assuming correctness of the guarded literals in Lattices:*

- *If LB-CEGAR outputs* **Safe***, the program* $P$ *is correct with respect to* $t$*.*
- *If LB-CEGAR outputs* **Unsafe** *with an accompanying CEX, the CEX demonstrates an execution of* $P$ *that falsifies* $t$*.*
- *If LB-CEGAR outputs* **Unknown***, the current theory and the set of guarded literals are insufficient to produce a conclusive result.*

*Proof.* Once *checkRealCE* is true with any set of guarded literals, the condition in line 22 is *true*, and the algorithm terminates. After exiting the loop in lines 8-22, since *checkRealCE* is true, the condition in line 27 holds, and LB-CEGAR

outputs **Unsafe** with the real counterexample, which is the counterexample it used last in the condition in line 22.

LB-CEGAR outputs **Safe** only if the conditions in line 24 holds, without lattices if the first query is **UNSAT** (in line 3) or with lattices if the last query is **UNSAT** (in line 20) and the condition for termination holds while using the frontiers of all lattices (line 22, when *termination* is $true$ and $result = true$). The first case is true in general in bounded model checking. We prove the second case where we use lattices.

The last query (line 19) just before violating the condition in line 22 is a conjunction of $\varphi$ with guarded literals from all elements of frontiers of all lattices in *Lattices*. By Lemma 1, we know that the union of *assume* statements of elements in the frontier captures the whole input domain of each of the functions of the lattices in *Lattices*.

Therefore, if no satisfying assignment has been found with all frontiers of lattices in *Lattices*, there is no satisfying assignment in the input domain of functions for which we added guarded literals (i.e., those which have a lattice in *Lattices*) in the unwound program $P$. Since there was no satisfying assignment for $\varphi$ in line 3, then the program is indeed safe with respect to the given bound and a property $t$.

Once LB-CEGAR cannot find a counterexample nor a set of guarded literals that refines $\varphi$ (that is *checkRealCE* is $false$, $\chi = \chi'$, or *termination* is $true$ but *result* is **SAT**) then none of the conditions in lines 24 and 27 holds and LB-CEGAR outputs **Unknown** (line 30). □

We observe that, while the worst-case complexity of LB-CEGAR is exponential in the number of lattices, in practice the algorithm is very efficient, as we show in Sec. 6.4.2. This is partly due to the *incrementality* of the calls to the SMT solver,

as the formula $\varphi$ representing $P \wedge \neg t$ stays the same for all iterations, and the next element $e'$ differs from the current element $e$ of the lattice only slightly. Another reason for the significantly lower complexity in practice is that our implementation of LB-CEGAR includes several heuristics, which we describe in the next section. The heuristics do not alter the correctness of the algorithm.

### Termination of LB-CEGAR algorithm

We prove here two additional lemmas required for the proof of Theorem 1.

The bound of each lattice $L$ in *Lattices* is as follows.

**Lemma 2.** *The bound of a lattice* $|L|$ *is* $(s+1) \times \binom{s}{\lfloor \frac{s}{2} \rfloor}$*, where* $s = |F|$ *and* $F$ *is a set of guarded literals of a function* $f$*.*

*Proof.* The lattice $L$ is constructed as a subset lattice form a set $F$ of guarded literals. The *height* of $L$ is at most $(s+1)$ by construction. The *width* of $L$ is bounded by $\binom{s}{\lfloor \frac{s}{2} \rfloor}$, following from Sperner's theorem [And87] that states that the width of the inclusion order on a power set is $\binom{s}{\lfloor \frac{s}{2} \rfloor}$, where $s$ is the size of the set. By using the height and the bound on the width of $L$, we get that the bound of the size of the lattice is:

$$|L| \leq (s+1) \times \binom{s}{\lfloor \frac{s}{2} \rfloor}$$

as required. □

**Lemma 3.** *The LB-CEGAR algorithm terminates.*

*Proof.* Sine $P$ is a bounded loop-free program then the set *Lattices* is finite. From Lemma 2 we know that the size of each lattice in *Lattices* is finite (since the size of each lattice is bounded).

Each iteration of the LB-CEGAR algorithm either invokes the sub-procedure $traverse_{UNSAT}$ or the sub-procedure $traverse_{SAT}$. In both cases, we traverse to

an element which **we yet visit**: in the former case, $traverse_{UNSAT}$ performs a traversal to an element whose set of literals corresponds to a different subdomain of the input, and in the latter case, $traverse_{SAT}$ performs a traversal to an element with a bigger set of guarded literals that refines the current counterexample; if $|Lattices| > 1$, we label a combination of elements from *Lattices* as visited instead of labelling an element.

The number of the lattices is finite; the size of each lattice is bounded and hence finite too. Accordingly, the number of times that LB-CEGAR traverses an element that has not yet been visited in each of the lattices is bounded and hence finite. Once LB-CEGAR has no new element to visit (but yet found a counterexample nor found guarded literals that suffice to prove correctness for all subdomains), then $\chi$ is equal to $\chi'$ and the LB-CEGAR algorithm terminates (exits the loop in line 22 and outputs a result in lines 24-30). □

## 6.4 Implementation and Evaluation

In this section, we present additional details of the implementation of the LB-CEGAR algorithm in HiFrog on top of the prototype implementation presented in Sec. 5.6.1, with an evaluation of the new implementation.

### 6.4.1 Implementation of the LB-CEGAR algorithm

The algorithms were implemented on top of the SMT-based function summarisation bounded model checker HiFrog (Chap. 4, [AAC+17]) with the OpenSMT2 SMT solver [HMAS16] and the Z3 SMT solver [DMB08b, DMP13]. The details of our initial implementation are described in Chap. 5. Here we describe the extension of the implementation to support the LB-CEGAR algorithm.

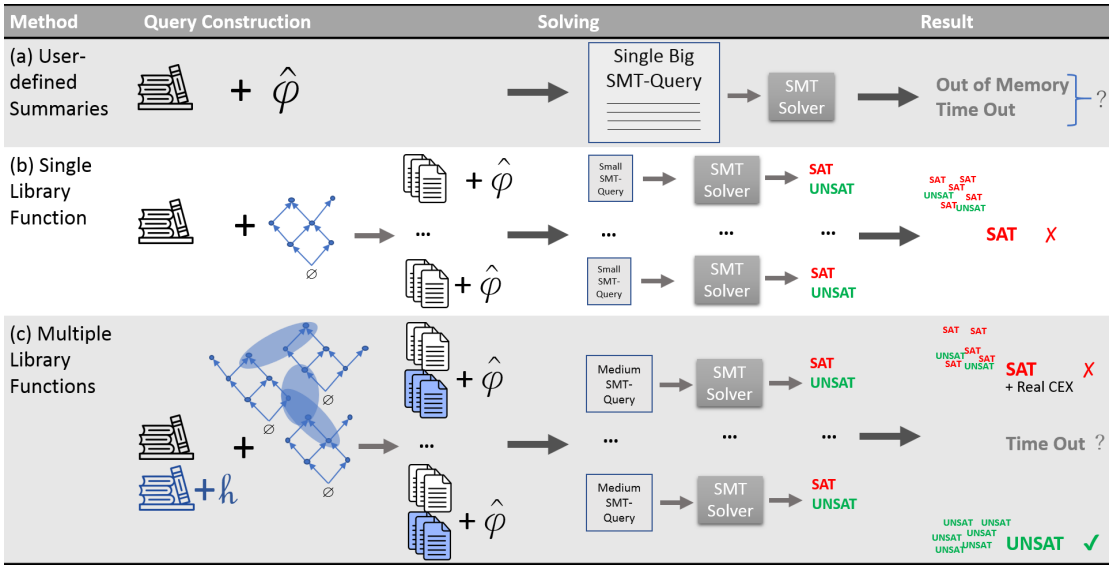


**Figure 6.2:** LB-CEGAR for program $P$ with several library functions (c) in comparison with BMC with UDS (a) and BMC with the initial approach LB-CEGAR-BMC (b).

Fig. 6.2 presents a high-level view of the implementation of LB-CEGAR in HiFrog and a comparison between the implementation as a flat (non-hierarchical) set of user-defined summaries, our prototype implementation with one function (LB-CEGAR-BMC), and the current implementation of the general algorithm (LB-CEGAR).

### Pre-processing stage

We constructed two lattices for sin and cos functions via a set of BASH scripts (see Chap. 5) for the evaluation of our approach. The guarded literals were imported from the raw data of Coq proof assistant [The19a] and Wikipedia [Lis19, Tri19] and translated to SMT summaries. The definitions of constants (e.g., $\pi$ from *math.h*) and trigonometric tables (values of the trigonometric functions for $x = c \cdot \pi$, for some $c \in \mathbb{N}$) were added to the set of guarded literals manually. The final set consisted of 80 guarded literals and was used to construct the meet semi-

lattices for $\sin x$ and $\cos x$ functions. Textual files of these meet semi-lattices are available at [Git19b, HiF19].

### Implementation in HiFrog

The implementation of LB-CEGAR uploaded only the set of guarded literals in the current element of the lattice for better performance. In LB-CEGAR, if the current element is insufficient for solving the formula (that is, the satisfying assignments produced by the SMT solver do not induce concrete counterexamples), the algorithm traverses the lattice to a higher element. In the implementation, this was translated as adding and removing some subsets of guarded literals. It is clear that the new formula only differed from the one in the previous iteration by a subset of guarded literals. The implementation exploited this fact by using the SMT solver in an *incremental* mode.

We extended the support for incremental solving in HiFrog, adding non-, semi-, and full-incremental solving modes, to support different degrees of incrementality (e.g., the semi-incremental solving mode allows only $push()$ calls). With this support, the implementation modified only a single query from one iteration to the next, which was less costly than re-writing the whole formula.

### Heuristics

We implemented the following heuristics to improve the complexity of lattice traversal in LB-CEGAR. None of these heuristics changed the worst-case running time complexity, but our experiments showed that they were beneficial on programs in our benchmark set.

- The choice of the successor in the sub-procedure $traverse_{SAT}(Lattices)$ was done based on the current spurious counterexample $CEX$, similarly to the

traditional CEGAR. We identified the location in the code where the abstract counterexample deviated from a concrete execution and used this information to guide the lattice traversal to the element that refined this particular location (if such an element existed).

- The 'frontier of unsatisfiability', that is, a frontier of a lattice that results in **UNSAT** for each element of this frontier, was computed once per lattice and was fixed. While in theory, it is possible that the current frontier of a lattice $L_1$ results in **UNSAT** when combined with an element $e$ of a lattice $L_2$, but not with an element $e'$ of $L_2$, in practice such cases were rare. There is an option to output **Unknown** if the set of frontiers computed gradually does not result in **UNSAT**, thus potentially increasing the number of cases, where LB-CEGAR outputs an inconclusive result. In our experiments, this heuristic did not lead to an increase in the number of inconclusive results.

- For lattices representing different occurrences of a function $f$ in $P$ which occur in a loop, we traversed these lattices simultaneously. The motivation for the 'coordinated' traversal was that all loop iterations, except, perhaps, for the last one, are similar, and hence there is a high probability that the same set of guarded literals would fit all these occurrences.

- We dynamically interpreted non-deterministic expressions in a guarded literal (an equation or inequality of a library function). Non-deterministic expressions were those expressions that we could not model in the current level of abstraction. In the current implementation, we interpreted these expressions by using other expressions in the first-order formula of the current encoding of the problem to guess the possible values of these non-deterministic expressions.

The main difference from Chap. 5 was that we treated non-deterministic expressions in UDS as we treated uninterpreted functions, without any attempt to narrow to a subdomain of values. In this chapter, we used the mechanism of function summaries merely for loading the lattice into our tool, but we interpreted the guarded literals concerning other library function calls in the program.

### 6.4.2 Experimental results

For the evaluation of LB-CEGAR, we constructed two lattices for sin and cos functions with 40 and 38 guarded literals, respectively. The validation test for these expressions contains a set of 144 benchmarks in C with a total of 365 assert statements. The scripts for the lattice construction, the benchmarks for the validation test, and the results of the validation test are available at [Git19a, HiF19].

The set of benchmarks contained a mix of our crafted benchmarks, programs from the software verification competition SV-COMP [Com18], and HiFrog benchmarks (Chap. 4, [AAC+17]), with a total of 141 C programs with at least one library function call, containing in total 194 calls for sin and 179 calls for cos, with 279 claims (127 **SAT** and 152 **UNSAT**). In 42 benchmarks, the library function is called at least 4 times, and in 8 benchmarks, the library function call is in a loop. The crafted benchmarks either assert known properties of trigonometric functions or contain a small part of code that is typical to kinematic problems, mainly examining the ability of verifying code with multiplication between two library function calls; e.g., $\cos\phi \times \sin\theta$.

To model a program with its property, we either used the quantifier-free SMT theory for equality logic with uninterpreted functions (EUF) with a semi-incremental solving mode in the OpenSMT2 SMT solver [HMAS16] or the

quanti- fier-free SMT theories for linear arithmetics (LA) with EUF with an incremental solving mode in the Z3 SMT solver [DMB08b]. For the CEGAR-style check of counterexamples in the sub-procedure *checkRealCE* in Sec. 6.3.2, we used the quantifier-free SMT theory for non-linear real arithmetic (NRA).

The default parameters of the LB-CEGAR algorithm include the use of all the heuristics in Sec. 6.4.1. In the evaluation, we used the default parameters and thus used all of the four heuristics; see [HiF19] for more details regarding these parameters.

The experiments were performed on a virtual machine (VM) with Ubuntu 16.04 Linux system, single-core, 8GB RAM; the VM runs on a machine with 4-Intel i7-6600U CPUs clocked at 2.60GHz. The experimental results, the benchmarks, and the source code, are available at [Git19a, Git19b, HiF19].

### Evaluation of LB-CEGAR with real arithmetic

**Figure 6.3** presents the comparison of LB-CEGAR with (i) CBMC version 5.10 [cbm19], (ii) HiFrog (Chap. 4, [AAC+17]), and (iii) LB-CEGAR-BMC our prototype implementation described in Chap. 5 supporting one library function at a time.

The total number of **solved instances** is the blue bar and the orange bar, for **Safe** and **Unsafe** instances respectively. The total number (as a negative number) of **unsolved instances** is the gray bar and the yellow bar, for **SAT** instances that are classified as **Unknown** (or **SAT** without a counterexample), and for the instances that timed out (TO) or were out-of-memory (OM), with the time-out set to 4000s and out-of-memory set to 3GB, respectively.

The four different colours of the bars are consistent across all six charts. Each

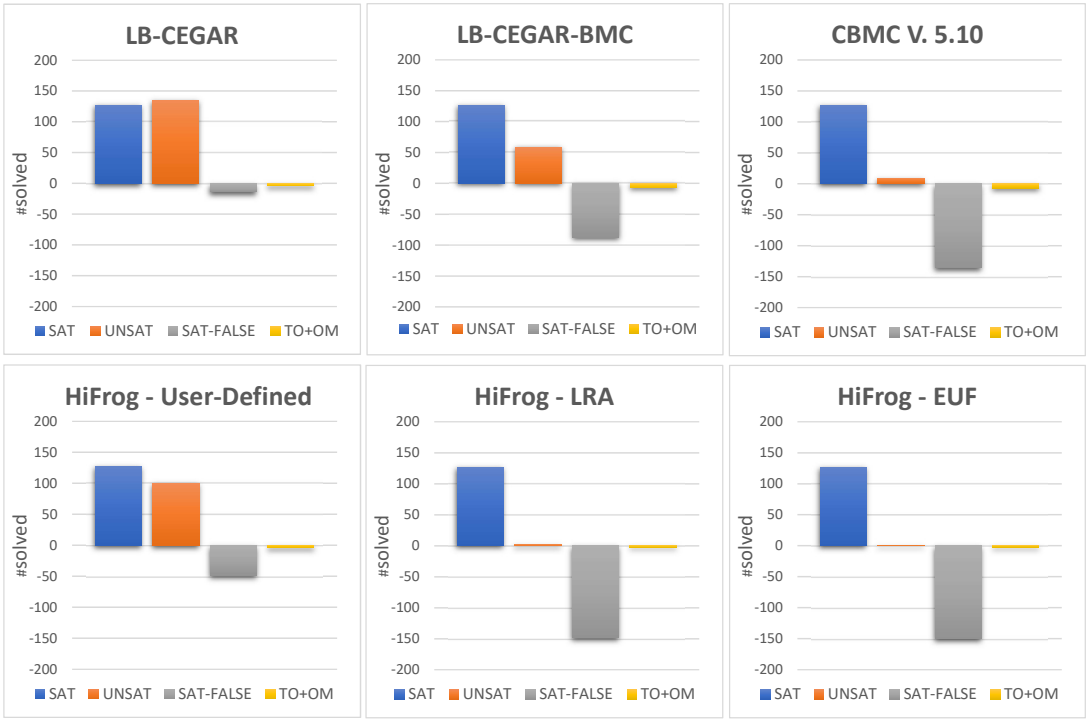


**Figure 6.3:** Comparison of the number (#) of solved claims with different approaches and a set of trigonometric benchmarks in C.

chart represents the total solved instances for a particular tool or a variant of a tool. The tools at the clockwise order are,

- LB-CEGAR with UFLRA (top-left).
- LB-CEGAR-BMC (Chap. 5) with UFLRA and a single lattice (top-middle).
- CBMC [CKL04] (top-right).
- HiFrog (Chap. 4) with user defined summaries with LRA (bottom-left), with LRA (bottom-middle), and with EUF (bottom-right).

Note that, HiFrog used either EUF **or** LRA (the quantifier-free SMT theory

for linear real arithmetic). All approaches with summaries used EUF with LRA (UFLRA).

Verification with function summaries in HiFrog avoids processing the single static assignment (SSA) expression of the original function, and only uses SMT summaries (Chap. 4). The SMT summaries in the experimental section here[3] were an over-approximation of library functions and contained no computation of the actual function nor of its Taylor series approximations, at any stage. Hence, complicated benchmarks, which contained library function calls in a loop or a non-linear expression, were more likely to be successfully verified with summary-based approaches than with classical approaches. These (unlike our method) usually required computation (up to some precision) of an approximation of the function and ran out of resources eventually. HiFrog with user-defined summaries used $\sim 80$ equations of trigonometric properties, loaded at once as unstructured data and solved a total of 227 instances (HiFrog - User-Defined, bottom-left, Fig. 6.3). HiFrog with a single lattice used $\sim 40$ equations of trigonometric properties, and solved a total of 185 instances (LB-CEGAR-BMC, single lattice, top-middle, Fig. 6.3).

LB-CEGAR, two lattices (*sin* and *cos*), constructed from a set of $\sim 80$ equations of trigonometric properties (LB-CEGAR, top-left, Fig. 6.3), outperformed both variants of HiFrog and had the highest overall number of *solved instances*: over **260** instances.

Connecting lattices of different functions in LB-CEGAR algorithm allowed our approach to verify the highest number of **Safe** instances, on many on which other tools failed (including HiFrog-User-Defined and LB-CEGAR-BMC with sin-

[3]There is no limitation on using a summary with the actual definition of a function with our method in general, as we have shown in [EAH+18] for Modulo function. However, the generalised technique in this chapter designed to perform well, even without such a summary, as schematically shown in Fig. 6.2 (c).

gle lattice). In comparison with LB-CEGAR (with 261 solved instances, out of 279), other tools that did not use summaries or insights on trigonometric functions, managed to solve at most 136 instances (CBMC V. 5. 10, HiFrog-LRA, and HiFrog-EUF, in Fig. 6.3, at top-right, bottom-middle, and bottom-right, respectively), mainly because of the use of non-deterministic variables to represent trigonometric functions.

**Evaluation of LB-CEGAR with different parameters of HiFrog**

Table 6.1 presents a full comparison of our implementation of the LB-CEGAR algorithm with other tools. The comparison was performed using various parameters of HiFrog, even-though LRA with EUF is the most suitable theory-combination for trigonometric functions, based on our experience.

The physical files of SMT summaries for LRA and EUF (UFLRA) were the same; HiFrog read an SMT summary file differently according to the theory in use. The SMT summaries for the quantifier-free SMT theory for linear integer arithmetic (LIA) were different to prevent any conversions to real arithmetic (e.g., *to_real* token) in the SMT query where all guards and literals were modelled via LIA with EUF (UFLIA).

In Table 6.1, the verification results of LB-CEGAR appear in white and are compared to CBMC [CKL04] and HiFrog (Chap. 4 and Chap. 5) appeared in grey and dark grey. The best results underlined and marked in light-yellow. The symbol # stands for the number of instances solved (first two lines) or unsolved (last two lines). Unsolved instances are false-negative results (FN-SAT), inconclusive (also marked as FN-SAT), or time-out or out-of-memory ($TO + OM$).

The description of each column in Table 6.1 is as follows.

- The **white** columns in Table 6.1 (3 columns, **LB-CEGAR col.**) contain

**Table 6.1:** Comparison of LB-CEGAR in HiFrog with different parameters with CBMC and HiFrog (comparison is on the number of solved and unsolved instances, with the time-out (TO) set to 4,000s and out-of-memory (OM) set to 3GB).

| | | **LB-CEGAR** | | | **LB-CEGAR-BMC** | | | **HiFrog UDS** | | | **HiFrog** | | | **CBMC** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | LRA | LIA | EUF | LRA | LIA | EUF | LRA | LIA | EUF | LRA | LIA | EUF | |
| #Solved | UNSAT | **134** | 42 | 8 | 59 | 13 | 6 | 100 | 24 | 9 | 2 | 3 | 1 | 9 |
| | SAT | **127** | 126 | 126 | 126 | 125 | 126 | **127** | 126 | 126 | **127** | **127** | **127** | **127** |
| #Failed | FN | **14** | 104 | 136 | 87 | 133 | 139 | 48 | 122 | 136 | 148 | 147 | 149 | 136 |
| | TO+OM | 4 | 7 | 9 | 7 | 8 | 8 | 4 | 7 | 8 | **2** | **2** | **2** | 7 |

the results LB-CEGAR presented in this chapter along with the sets of heuristics presented in Sec. 6.4.1 and different sets of parameters (with the theory being EUF, UFLRA ,or UFLIA in the columns EUF, LIA and LRA respectively), against other tools in the grey scale columns.

- The **grey scale** columns in Table 6.1 (10 columns from the end) contain the results of other tools: LB-CEGAR-BMC with a single lattice (Chap. 5) in **LB-CEGAR-BMC col.**, HiFrog with a large set of user defined summaries (Chap. 4) in **HiFrog UDS col.**, and HiFrog (Chap. 4, [AAC+17]) and CBMC version 5.10 [cbm19] in the **most right columns**.

  The evaluation of HiFrog in grey-scale is with a single lattice, and with and without user defined summaries, using: EUF, LRA, LIA, LRA with EUF (UFLRA), LIA with EUF (UFLIA), each of which is a different column in Table 6.1 (with the theory being EUF, LRA or LIA).

The variant of HiFrog with theory refinement is omitted from the comparison, as its current implementation does not support trigonometric functions.

For the LB-CEGAR approach for verification of programs with multiple trigonometric functions, the best setting was with LRA with EUF (LB-CEGAR, LRA col.), which had the highest overall number of *solved instances* over **260** instances, and performed almost as well as HiFrog without any summary (ran out of resources 4 times versus 2 times HiFrog did). The other two configurations of parameters we tried with the LB-CEGAR approach for programs with multiple library functions, were EUF (LB-CEGAR, EUF col.) and LIA with EUF (LB-CEGAR, LIA col.). While EUF performed poorly in general, LIA with EUF has shown limited potential in solving instances that required real arithmetic, which indicates the possibility of applying this method for code with library functions

over significantly different input domains, that is, code with both continuous and discontinuous functions.

The comparison with LB-CEGAR-BMC with a single lattice (Chap. 5) (LB-CEGAR-BMC, LRA col.) used either a meet semi-lattice for sin function or for cos function per benchmark, which led to poor performance when both lattices were required; however, perhaps unsurprisingly, this did not result in poor performance when a single lattice was sufficient to prove the safety of a claim (59 instances, LB-CEGAR-BMC, LRA col.). HiFrog with user-defined summaries (HiFrog UDS, LRA col.) could not solve around 50 safe instances that required a wider context regarding other library functions, for one or more expressions with a library function call.

## 6.5 Related Work

In this section, we discuss the related work to the lattice construction algorithm, and the LB-CEGAR algorithms for efficient verification of software with library functions (Chap. 5-6), including the use of mathematical properties, lattices, function summaries, SMT solvers, and other techniques for this purpose.

Lattices are a useful mathematical structure in understanding the relationships between different abstractions and have been widely applied in a program solving with Craig interpolation [Cra57]. For instance, [RS13] presented a semantic solver-independent framework for systematically exploring interpolant lattices using the notion of interpolation abstraction; the framework tried to find the right level of abstraction (or description) of an interpolant of a problem (a chuck of code). A lattice-based system for interpolation in propositional structures was presented in [DKPW10], extended to consider size optimisation techniques in

the context of function summaries in [RAF+13, AFHS15], and further extended to partial variable assignments in [JAF+16]. Similar lattice-based reasoning has also been extended to interpolation in first-order logic with other SMT, including the equality logic with uninterpreted functions [AHAS17], and linear real arithmetic [AHS17]. The approach presented in this chapter differs from the above in that we did not rely on interpolation and worked in tight integration with the model checker.

Computationally inexpensive theories can be used to over-approximate complex problems. These approaches have been used in solving equations on non-linear real arithmetic and transcendental functions based on linear real arithmetic and equality logic with uninterpreted functions [KIY16, CGI+17a, CGI+17b, CGI+18b], as well as on scaling up bit-vector solving [HCR+16, AAC+17, HAE+17, BT19]. Our work in Chap. 5 and Chap. 6 can be seen as a generalisation of these approaches as we supported inclusion of lemmas from more descriptive logics to increase the expressiveness of computationally lighter logics.

The problem of verification of programs with transcendental functions and, in particular, trigonometric functions have been addressed by several verification tools. iSAT3 [FHT+07] handled the problem via interval propagation by refining the computed interval bounds, and dReal [GKC13] also used interval propagation with $\delta$-satisfiability but with user-specified precision where $\delta$ is associated with the numerical error. Coq interval [Mel12] and Gappa [DDLM11] applied interval propagation with Taylor series, while MathSAT5 [CGSS13, CGI+17b, CGI+18b, IGCS19] applied Taylor series with a partial set of trigonometric properties. CVC4 [RTJB17, BT19] and MathSAT5 [CGI+17a, CGI+17b, CGI+18a, CGI+18b, IGCS19] adopted incremental linearisation (as defined in MathSAT5) for solving non-linear arithmetic prob-

lems in general. [Har00] addressed the problem for hardware verification. In contrast to these approaches, our algorithm did not require non-linear arithmetic or a calculation of Taylor series, which is computationally expensive for large programs.

Lattices and posets have been used in abstract interpretation [CC77] to model a sound approximation of the semantics of code, where completeness and partial completeness [CC79, Cou00, GRS00, GQ01] referred to the no loss of precision during the approximation of the semantics of code. Giacobazzi et al. [GRS00, GQ01] presented the notation of backward and forward completeness and showed the connection between iteratively computing the backward (forward)-complete shell to the general CEGAR framework [CGJ+03]. The completeness of their algorithm depended on the properties of the abstraction, while our algorithm had no such requirements.

The idea of applying SMT solvers with abstract interpretation for program verification and the formalism of the unified approach with sufficient conditions for completeness were presented in [CCM12]. A generic abstract satisfaction framework [DHK14] with an efficient implementation [Hal13] applied the language of abstract interpretation into SMT solvers and presented a simplified formalism with fix-point completeness. These approaches applied abstract domains in SMT solvers to deal with often difficult to model program properties. For trigonometric functions where the implementation is usually based on Taylor approximations, the program properties and the properties of trigonometric functions in mathematics differ substantially.

Program slicing [Wei81] aims to reduce the size of a program by decomposing the program into program slices according to a slicing criterion. In backward slicing [Wei81, Sil12] the program slices were defined as the executable subset

of statements with the same behaviour while slicing out any statement with no effect on the variables and the control flow of the property under test. Attempts to reduce further the size of a slice by aggressive slicing techniques (e.g., by aggressive slicing [SFB07, KSK15]) remind the use of an *assume* statement of guarded literals in the meet semi-lattice in our work. The similarity of the techniques was in reducing the size of SMT queries by decomposing the expressions into meaningful subsets each of which referred to a set of values of variables of the expression (in our case, the library function call). *Thin slicing* [SFB07] kept only data-dependent statements relevant to the property while removing control flow statements. *Value slicing* [KSK15] similarly to thin slicing kept statements affecting values of the variables in the property while bringing back conditional statements that are value-impacting relevant to the property.

Interesting work on combining theorem provers with SMT solvers included the SMTCoq system [EMT+17]. Our work in Chap. 5 and Chap. 6 used equations and inequalities of properties of mathematical functions from the Coq library, but differed from SMTCoq in that we imported the equations and inequalities of the properties directly to the SMT solver instead of giving the SMT solver to Coq.

## 6.6 Conclusions and Future Work

We presented a new algorithm LB-CEGAR that is used for verification of programs with library functions, for which a number of equations, some of which are instrumental for verification of these programs, exist in external sources (the mathematical library, the Coq proof assistant, etc.). The main idea of the algorithm is to organize the equations in subset lattices, and to replace the traditional CEGAR refinement loop with lattice traversal. The algorithm is general in the

sense that it allows several occurrences of the same library function and/or several different library functions, some of which depend on each other, in the same program. While the theoretical worst-case complexity of LB-CEGAR is high due to an exponential number of combinations of elements of different lattices, our experimental results show that the algorithm is very efficient in practice and outperforms state-of-the-art model checking tools on benchmarks with trigonometric functions.

We view the programs with trigonometric functions as the primary domain of application of LB-CEGAR. In the future, we plan to explore the domain of verification of programs describing robots' movements and kinematics in general.

# Chapter 7

# Theory Refinement for Program Verification

Recent progress in automated formal verification is to a large degree due to the development of constraint languages that are sufficiently light-weight for reasoning but still expressive enough to prove properties of programs. Satisfiability modulo theories (SMT) solvers implement efficient decision procedures but offer little direct support for adapting the constraint language to the task at hand.

*Theory refinement* is a new approach that modularly adjusts the modelling precision based on the properties being verified through the use of a combination of theories. We implement the approach using an augmented version of the theory of bit-vectors and uninterpreted functions capable of directly injecting non-clausal refinements to the inherent Boolean structure of SMT. In our comparison to a state-of-the-art model checker, our prototype implementation is in general competitive, being several orders of magnitudes faster on some instances that are challenging for flattening, while computing models that are significantly more succinct.

## 7.1 Introduction

The satisfiability modulo theories (SMT) [DNS05] reasoning framework is currently one of the most successful approaches to verifying software in a scalable way. The approach is based on modelling the software and its specifications in propositional logic, while expressing domain-specific knowledge with first-order theories connected to the logic through equalities. Once a satisfying assignment is found for the propositional model, its consistency is queried as equalities from the theory solvers, which, in case of inconsistency, provide an explanation as a propositional clause. Successful verification of software relies on finding a model that is expressive enough to capture software behaviour relevant to correctness, while sufficiently high-level to prevent reasoning from becoming prohibitively expensive. Since in general more precise theories are both more expensive computationally and potentially distracting for the automatic reasoning, finding such a balance is a non-trivial task.

We introduce *theory refinement*, a counterexample-guided abstraction refinement (CEGAR) [CGJ+00, CGJ+03] approach for modelling software modularly using theories that are partially ordered with respect to their precision. Our main contribution is the process of gradually encoding a program using the most precise theory only for a critical subset of all program statements, while keeping lower precision for the rest of the statements. The critical subset of theories is identified based on counterexamples, and theories of different precision are bound to each other through special identities. We study several automatic heuristics for guiding the encoding and provide also a manual encoding option. We apply theory refinement on verification of safety properties of software through bounded model checking. However, we believe that the technique is applicable in most

verification techniques where higher-level information is available on the problem structure. This includes model checking [BCCZ99] and upgrade checking [FSS13], $k$-induction [McM05], the IC3 algorithm [Bra11], and generation of inductive invariants [GBM14]. We show that the modular composition of the theories preferring lower precision can be used to both obtain speed-up in solving and identifying statements whose precise semantics do not affect the program safety, providing the model checker with cleaner proofs.

Many SMT solvers use over-approximation through theories as a means of speeding up solving. For instance [BCF$^+$07, BB09, HBJ$^+$14] organizes the theory solvers into layers that solve problems represented in BV. The query is first given to fast and less precise theory solvers, and only passed on to the exact solver if previous layers fail to show unsatisfiability. In contrast to low-level SMT solving, the work in this chapter studies how to automatically identify statements whose exact semantics can be ignored in model checking. This shift of view point has several advantages: (i) the approach can be used both to obtain speed-up in solving, and as a means for synthesis and finding fix-points for transition relations; (ii) the guidance from the source code allows the use of more powerful heuristics for choosing which statements should remain abstract; and (iii) the refinement takes place on the level of the program, not at the level of the theory query, an approach potentially more natural from the point of view of the semantics of the program.

We present theory refinement with two new theories called *uninterpreted functions for programs* (UFP) and *bit vectors for programs* (BVP) that are based on the theories of quantifier-free uninterpreted functions with equality (EUF), and bit vectors (BV), respectively. The two theories were chosen since they represent two natural extremes in precision and are commonly used in the layered solver

approach (see, e.g., [HBJ+14]). In addition to the functionality of EUF, UFP provides interpretations for constants, conversion of abstract values to concrete values, and commutativity for uninterpreted functions when applicable. The key difference in BVP compared to BV is that BVP is capable of directly injecting non-clausal refinements, modelling the program statements bit-precisely, to the inherent Boolean structure maintained in the SMT solver.

We implemented theory refinement on the SMT solver OpenSMT2 [HMAS16] and the bounded model checker HiFrog (Chap. 4, [AAC+17]) supporting a subset of the C language. We report promising results both with respect to speed and the amount of refined program statements on both instances from a software verification competition and our own regression test suite. We demonstrate that the approach has a potential of several orders of magnitude of improvement over the approach based solely on flattened bit-vectors, as implemented in the state-of-the-art tool CBMC and in our own tool. The implementation and the benchmarks are available at [HiF17a].

## 7.2 Combination of Theories in Theory Refinement

This section fixes a notation for describing instances of the safety problem using SMT and provides two communicating theories for solving the safety problem. The goal of the presentation is to clarify how the modelling works in the SMT framework, placing particular emphasis on the use of symbols and their semantics.

In modelling programs, we consider sets of quantifier-free symbolic statements of the form $x = t$, where $x$ is a variable, and $t$ is a term. This form essentially corresponds to the Single static assignment (SSA) form for loop-free programs

**Table 7.1:** The functions used in the encoding in this chapter (unsigned and signed sum coincide).

| **Functions** | | **Descriptions** |
|---|---|---|
| | Logical functions | |
| `&&` , `\|\|` | $Sb \times Sb \to Sb$ | Logical and, or |
| `!` | $Sb \to Sb$ | Logical not |
| | Non-logical functions | |
| `+` , $*_u$ , $*_s$ , $/_u$ , $/_s$ | $Sz \times Sz \to Sz$ | Sum, unsigned and signed product and division |
| $\%_u$ , $\%_s$ | $Sz \times Sz \to Sz$ | Unsigned and signed remainder |
| $\ll, \gg_a, \gg_l$ | $Sz \times Sz \to Sz$ | left shift, arithmetic and logical right shift |
| `&` , `\|` , `^` | $Sz \times Sz \to Sz$ | Bitwise and, or, exclusive or |
| $\sim$ : | $Sz \to Sz$ | bitwise complement |
| $\leq_s$ , $\leq_u$ , $<_s$ , $<_u$ , $\geq_s$ , $\geq_u$ , $>_s$ , $>_u$ | $Sz \times Sz \to Sb$ | Signed and unsigned less than or equal to and greater than or equal to |

(see Chap. 2, Sec. 2.4.3). The symbolic statements are defined over a sort of bounded integers $Sz$ and a Boolean sort $Sb = \{\top_l, \bot_l\}$; we distinguish between these sorts and, for instance, the sorts of integers $\mathbb{Z}$ and Booleans $\mathbb{B}$ to clarify the difference between this *symbolic encoding* (hence the $S$) and the representation used by an SMT solver. Table 7.1 lists the non-variable functions we consider in our encoding. Note that unlike some programming languages, including C and C++, we do not allow the encodings to interpret terms from $Sz$ as terms from $Sb$ or vice versa. We distinguish between the functions defined over the sort $Sb$ and those defined over $Sz$, calling the former logical functions and the latter non-logical functions. The control-flow structures, such as `if-then-else`s, are encoded using the functions `!` , `||` , and `&&` . For the purpose of this presentation

| | |
|---|---|
| | $\left(c^b =_{BVz} \left((a^b \,\%_u\, 2^b) + (b^b \,\%_u\, 2^b)\right) \,\%_u\, 2^b\right)_1 \wedge$ |
| $c = \left((a \,\%_u\, 2) + (b \,\%_u\, 2)\right) \,\%_u\, 2$ | $\left((c')^b =_{BVz} (a^b + b^b) \,\%_u\, 2^b\right)_1 \wedge$ |
| $c' = (a + b) \,\%_u\, 2$ | $\left(d^u = f^u *_u e^u *_u c^u\right) \wedge$ |
| $d = f *_u e *_u c$ | |
| $d' = e *_u f *_u c'$ | $\left((d')^u = e^u *_u f^u *_u (c')^u\right) \wedge$ |
| | $\left(c^u = (c')^u\right) \leftrightarrow \left(\left(c_1^b \leftrightarrow (c')_1^b\right) \wedge \ldots \wedge \left(c_{bw}^b \leftrightarrow (c')_{bw}^b\right)\right)$ |

**Figure 7.1:** (*Left*) sequence of statements and (*right*) the corresponding encoding in combined UFP and BVP (to be described in Sec. 7.2.3, on the left all the variables are of sort $Sz$, and $e$ and $f$ are unbound).

we assume that the encodings do not contain arrays and pointers[1]. Fig. 7.1 (*left*) shows an example sequence of statements that we use as a running example in the discussion of this section.

### 7.2.1 Bit vectors for programs

Our theory of bit vectors for programs (BVP) has a single sort $BVz^{bw}$ containing the integers representable in $bw \in \mathbb{N}$ bits. When the bit-width of the sort is clear from the context we simply write $BVz$ for the sort. Each BVP term $t$ of sort $BVz^{bw}$ is associated with the bits $t_1, \ldots, t_{bw}$ which are variables from the sort $\mathbb{B}$. The bits $t_1$ and $t_{bw}$ are called, respectively, the *least significant bit* and the *most significant bit* of $t$.

The BVP theory has two special constants $1^b$ and $0^b$. For the constant $0^b$, $0_i^b = \bot$, $1 \leq i \leq bw$. For the constant $1^b$, $1_1^b = \top$ and $1_i^b = \bot$ for $2 \leq i \leq bw$. The equality of BVP is $=_{BVz}$: $BVz \times BVz \to BVz$. The interpretation of the equality is that if $x =_{BVz} y$ holds, then the value of the equality term is $1^b$ and otherwise $0^b$. Finally, BVP has the functions defined in Table 7.1 with all sorts replaced

[1]We do support these in our implementation, but their results are treated non-deterministically, that is, as unbound variables from $Sz$.

by the sort $BVz$. For a term $t$, the Boolean functions determining the bits $t_i$ are computed through propositional flattening (see, e.g., [KS08]).

We encode a sequence of statements $P = \{x_1 = t_1, \ldots, x_n = t_n\}$ in BVP as follows. Each statement $x_i = t_i$ is converted to $|x_i|^b =_{BVz} |t_i|^b$, where the operator $|\cdot|^b$ is defined for a symbolic term $t$ recursively:

$$|t|^b \stackrel{\text{def}}{=} \begin{cases} x^b & \text{if } t \doteq x \text{ is a variable or a constant} \\ |x|^b \bowtie |y|^b & \text{if } t \doteq x \bowtie y \text{ where } \bowtie \text{ is a binary function,} \\ \circ|x|^b & \text{if } t \doteq \circ x \text{ where } \circ \text{ is a unary function} \end{cases} \tag{7.1}$$

where $a \doteq b$ denotes that the term $a$ matches the form of $b$. Conjunction of the least significant bits of encoded statements in $P$ defines its BVP-encoding $[P]^b$:

$$[P]^b \stackrel{\text{def}}{=} (|x_1|^b =_{BVz} |t_1|^b)_1 \wedge \ldots \wedge (|x_n|^b =_{BVz} |t_n|^b)_1 \tag{7.2}$$

We say that a safety property $t$ holds in program $P$ if and only if $[P]^b \wedge \neg[t]^b_1$ is unsatisfiable. Based on the definition we can see that the symbolic encoding in Fig. 7.1 satisfies the safety property $(d = d')$ due to properties of modular arithmetics. The BVP encoding is often inefficient due to the quadratic growth of the formula with respect to $bw$. However, in many cases, the bit-precise encoding of statements (e.g., $*_u$ in Fig. 7.1) are irrelevant to the safety property and can therefore be over-approximated. This motivates the use of less precise but more efficiently solvable encodings such as those based on uninterpreted functions.

### 7.2.2 Uninterpreted functions for programs

The logic UFP (Uninterpreted Functions for Programs) is the standard logic of quantifier-free uninterpreted functions having the Boolean sort $\mathbb{B}$, the standard

Boolean functions $op : \mathbb{B} \times \ldots \times \mathbb{B} \to \mathbb{B}$ where *op* is an operator such as $\vee, \wedge$, and $\neg$, and an unbounded number of variables. In addition, the logic is augmented with

- a sort *UFPn* of real or integer numbers;
- the functions listed in Table 7.1 treated as uninterpreted functions with the sorts *UFPn* and $\mathbb{B}$ instead of *Sz* and *Sb* respectively;
- commutativity of the functions `+`, $*_u$, $*_s$, `&`, and `|`; and
- the concept of constants beyond the Boolean $\top$ and $\bot$.

As usual, UFP also contains the equality function $=_S: T \times T \to \mathbb{B}$ for all sorts $T$. As in the symbolic encoding, also in UFP we differentiate between two types of functions: those with a return sort $\mathbb{B}$, and those with a return sort *UFPn*.

Given a sequence of statements $P = \{x_1 = t_1, \ldots, x_n = t_n\}$, we denote its encoding in UFP by $[P]^u \stackrel{\text{def}}{=} ([x_1]^u =_{T_1} [t_1]^u) \wedge \ldots \wedge ([x_n]^u =_{T_n} [t_n]^u)$, where $T_i$ is either *UFPn* or $\mathbb{B}$ depending on the related sort. The encoding operator $[\cdot]^u$ is defined as follows for a term $t$:

$$[t]^u \stackrel{\text{def}}{=} \begin{cases} x^u & \text{if } t \doteq x \text{ is a variable or a constant} \\ [x]^u \wedge [y]^u & \text{if } t \doteq x \texttt{ \&\& } y \\ [x]^u \vee [y]^u & \text{if } t \doteq x \texttt{ || } y \\ \neg[x]^u & \text{if } t \doteq \texttt{ ! } x \\ [x]^u \bowtie [y]^u & \text{if } t \doteq x \bowtie y \text{ where } \bowtie \text{ is a non-logical function.} \end{cases} \tag{7.3}$$

We distinguish between the notions of program safety in UFP and in BVP. In particular, we say that a safety property $t$ holds in program $P$ in UFP if and only if $[P]^u \wedge \neg[t]^u$ is unsatisfiable.

The program in Fig. 7.1 is safe with respect to the safety property $!(c = c') \,||\, (d = d')$ in UFP and therefore also in BVP. However, it is not safe in UFP with respect to the safety property $d = d'$ that is safe in BVP. For checking safety of programs in UFP we use a theory solver implementing a congruence closure algorithm [DNS05] that is modified to support constants and commutativity. The modifications are described in more detail in Sec. 7.4.1.

In the experiments in Chap. 4, we showed that safety of many programs can be established by interpreting the arithmetic functions as uninterpreted functions. In the next subsection we describe how the UFP logic and the BVP logic can be combined.

### 7.2.3 Combination of UFP and BVP

We present the theory refinement approach using a seamless integration of the UFP and BVP encoding, and therefore require a form of theory combination. However, unlike in conventional theory combination on bit vectors (see, e.g., [HBJ+14]), we do not need to consider bit-vectors as theories, but instead they are embedded directly to the Boolean structure of the SMT solver. The two theories UFP and BVP are combined using a *binding formula* defined as follows.

**Definition 2.** *Given a symbolic statement $t$, let $[t]^u$ and $[t]^b$ be its* UFP *and* BVP*-encodings respectively. If both $[t]^u$ and $[t]^b$ appear together in a formula, we say that $t$ is* bound*. Let $B$ be the set of all bound statements. The* binding formula *for $B$ (denoted $F_B$) is defined as*

$$F_B \stackrel{\text{def}}{=} \bigwedge_{t,t' \in B} ([t]^u = [t']^u) \leftrightarrow \Big(([t]^b_1 \leftrightarrow [t']^b_1) \wedge \ldots \wedge ([t]^b_{bw} \leftrightarrow [t']^b_{bw})\Big) \tag{7.4}$$

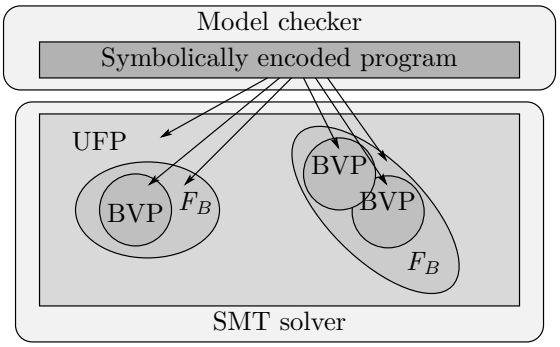


**Figure 7.2:** Symbolic encoding of program and the corresponding SMT formula.

Intuitively, the combination of the theories UFP and BVP with $F_B$ allow us to express an over-approximation of the symbolic encoding of a program. This is stated more formally in the following theorem.

**Theorem 3.** *Let $P$ be a program. Then $[P]^b \wedge F_B \models [P]^u$.*

*Proof.* (*sketch*) By simulation of executions in BVP: if there exist values $v_1^b, \ldots, v_n^b$ for the variables $x_1^b, \ldots, x_n^b$ in a term $[a = t]^b$ then the same values $v_1^u, \ldots, v_n^u$ satisfy the corresponding equality $[a]^u = [t]^u$. □

Fig. 7.2 shows the combined UFP and BVP encoding schematically. In the schematic example most of the program is encoded using UFP, while certain critical parts are encoded in BVP and made to communicate with the UFP encoding using the binding formula $F_B$. The symbolic encoding of a program is partitioned by the model checker into three parts: the UFP encoding, the BVP encoding, and the binding formula $F_B$. The conjunction of these is solved by the SMT solver. We describe how our method decides which critical statements to encode in BVP in the next section. Fig. 7.1 (*right*) describes a combination encoding of UFP and BVP together with the necessary binding formula for the running example.

**Algorithm 6:** The theory refinement algorithm

**input :** $P = \{(x_1 = t_1), \ldots, (x_n = t_n)\}$: a program, and $t$: a safety property
**output:** $\langle$**Safe**, $\bot\rangle$ or $\langle$**Unsafe**, $CEX^b\rangle$

```
1  For all 1 ≤ i ≤ n initialize ρ[x_i = t_i] ← [x_i = t_i]^u
2  ρ[t] ← [t]^u
3  F_B ← ⊤
4  while true do
5  |   Query ← ρ[x_1 = t_1] ∧ ... ∧ ρ[x_n = t_n] ∧ ¬ρ[t] ∧ F_B
6  |   ⟨result, CEX⟩ ← checkSAT(Query)
7  |   if result is UNSAT then
8  |   |   return ⟨Safe, ⊥⟩
9  |   end
10 |   CEX^b ← getValues(CEX)
11 |   foreach s ∈ P ∪ {t} s.t. ρ[s] ⊭ [s]^b do
12 |   |   ⟨result, _⟩ ← checkSAT([s]^b ∧ CEX^b)
13 |   |   if result is UNSAT then
14 |   |   |   ρ[s] ← refine^s(ρ[s])
15 |   |   |   F_B ← computeBinding(ρ)
16 |   |   |   break
17 |   |   end
18 |   end
19 |   if No s was refined at line 14 then
20 |   |   return ⟨Unsafe, CEX^b⟩
21 |   end
22 end
```

## 7.3 Counterexample-Guided Theory Refinement

This section provides an algorithm for verifying safety of programs by gradually refining the *precision* $\rho$ of the symbolic encoding from UFP to BVP in parts where satisfying truth assignments shows that it is necessary for soundness. Algorithm 6 describes the high-level idea. The algorithm takes as input a symbolically encoded problem $P$ and a safety property $t$, and returns either **Safe**, if $t$ holds in $P$, or **Unsafe** with a bit-precise counterexample if $t$ does not hold in $P$. During the execution the algorithm picks statements $s \in P \cup \{t\}$ and refines their approxi-

mations in $\rho$ until $\rho[s]$ is equivalent to $[s]^b$. Based on $\rho$, the algorithm constructs the binding formula $F_B$ sufficient to connect the UFP and BVP terms.

The safety of the program is tested at lines 5–9 using the current precision $\rho$ and the binding formula. If the check succeeds, the algorithm terminates at line 9. Otherwise, a satisfying truth assignment is extracted at line 10 and then used to refine $\rho$ at lines 11–18.

The need for refinement is checked for every statement $s$ with a precision $\rho[s]$ not equivalent to $[s]^b$. If the truth assignment $CEX^b$ is inconsistent with $[s]^b$ then $\rho[s]$ is refined to block the truth assignment. If at least one such replacement happens in the current iteration, the execution proceeds to line 5. In practice it is a good idea to refine several statements based on a single counterexample, as discussed in Sec. 7.4.2. If no refinement is done, the truth assignment corresponds to a counterexample and the algorithm terminates at line 20.

The algorithm uses four sub-procedures *checkSAT*, `getValues`, $\texttt{refine}^s$, and `computeBinding`. *checkSAT*$(F)$ determines the satisfiability of a formula $F$, `getValues`$(CEX)$ computes a BVP encoding of $CEX$ through substituting the abstract values from UFP with concrete BVP values. $\texttt{refine}^s(F)$ refines the statement $s$ with respect to the previous precision $F$, and `computeBinding`$(\rho)$ computes the binding formula using Def. 2. Below we give a definition for the `refine` procedure, while we discuss the other procedures in more detail in Sec. 7.4.1.

**Definition 3.** *The procedure* $\texttt{refine}^s(F)$ *returns an iterative refinement of the statement s of the symbolic encoding with respect to F, such that (i)* $\texttt{refine}^s(F) \models F$*, and (ii)* $\texttt{refine}^s$ *has a fix-point that is equivalent to* $[s]^b$ *and reachable in a finite number of applications of* $\texttt{refine}^s$*.*

While in the implementation discussed in Sect. 7.4.1 we use $\texttt{refine}^s(F) =$

$[s]^b \wedge [s]^u$, we want to point out the possibility of using interpolation-based methods (see, e.g., [AFHS15]) for the refinement.

**Theorem 4.** *Alg. 6 terminates in a finite number of steps.*

*Proof.* Assume that Alg. 6 does not terminate. Then there is a term in $P \cup \{t\}$ that can be refined an unbounded number of times before the fix-point equivalent to $[s]^b$ is reached, which contradicts Def. 3. □

**Theorem 5.** *Alg. 6 returns* **Unsafe** *if and only if the symbolic encoding $P$ has an execution violating the safety property $t$.*

*Proof.* The algorithm maintains the invariants

$$\begin{aligned} &\text{Inv1} \quad [x_1 = t_1]^b \wedge \ldots \wedge [x_n = t_n]^b \models \rho[x_1 = t_1] \wedge \ldots \wedge \rho[x_n = t_n] \\ &\text{Inv2} \quad [t]^b \models \rho[t] \end{aligned} \tag{7.5}$$

at line 14 by Def. 3 and Th. 3. Assume that the algorithm returns **Unsafe** but there is no execution violating the safety property $t$. Then there is a truth assignment $\sigma$ such that $\rho[x_1 = t_1] \wedge \ldots \wedge \rho[x_n = t_n] \wedge F_B$ is true and $\rho[t]$ is false. The truth assignment $\sigma$ must also satisfy $[x_1 = t_1]^b \wedge \ldots \wedge [x_n = t_n]^b$. By Inv2, if $\rho[t]$ is false also $[t]^b$ is false, hence contradicting the unsafety of $(P, t)$. Now assume the algorithm returns **Safe** but there is an execution of $P$ violating $t$. Then there is a truth assignment satisfying $[P]^b \wedge \neg[t]^b$. Since by Th. 3 both $[P]^b \wedge F_B \models \rho[x_1 = t_1] \wedge \ldots \wedge \rho[x_n = t_n]$ and $\neg[t]^b \wedge F_B \models \neg\rho[t]$, also the query on line 5 is satisfiable, contradicting the assumption. □

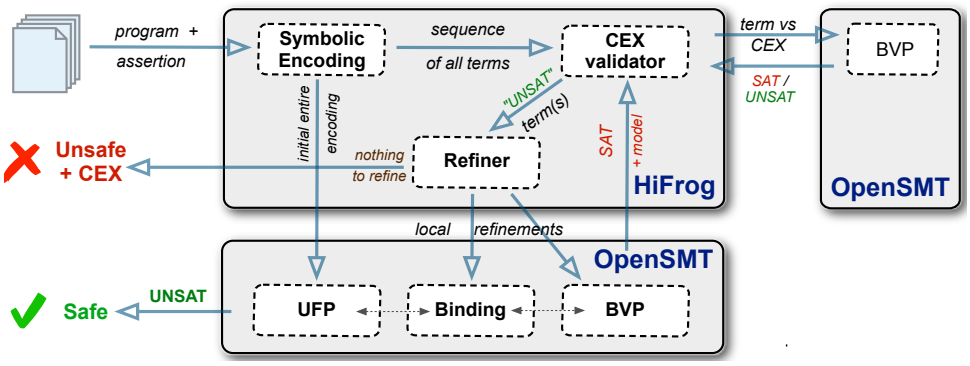


**Figure 7.3:** SMT-based model checking framework for theory refinement approach.

## 7.4 Implementation and Evaluation

In this section we present important details regarding the implementation of the functionality of the theory refinement algorithm in the solver and the model-checker and evaluating the efficiency of our algorithm.

### 7.4.1 Implementation of the theory refinement algorithm

This section describes the prototype implementation of the theory refinement algorithm. The algorithm was implemented on the SMT solver OpenSMT2 [HMAS16] and the bounded model checker HiFrog (Chap. 4, [AAC+17]). The overview of implementation includes the three main components and interactions between them is depicted in Fig. 7.3.

#### The solver for UFP

The UFP theory solver was based on the co-operation between a congruence closure algorithm, which maintained sets of equivalence classes and inequalities between the classes, and a SAT solver, which enforced a propositional structure describing the relations between the equalities. The equality graph (*egraph*) al-

gorithm represents the problem as an undirected graph to describe the relations between these equalities. We refer the reader to [DNS05] for the full description of the *egraph* algorithm that the UFP solver based on and [KS08] (Chap. 4) for the definition of *egraph* in decision procedures for EUF.

**Constants.** The original egraph algorithm did not support constants other than the Boolean $\top$ and $\bot$, but constants played often an important role in our benchmarks. The egraph algorithm can represent an inequality between two terms $t_1, t_2$ by asserting explicitly the inequality $t_1 \neq t_2$ over these terms. This representation grows quadratically in the number of constants and therefore is not scalable. We adopted a different strategy for representing the inequalities between constants. An equivalence class in the egraph algorithm was represented by a linked list binding together the terms in the same class. Each class was represented by a canonical term from the linked list. In the original algorithm of [DNS05], when two equivalence classes $a$ and $b$ were joined, the canonical term of the new class $a \cup b$ was the representative of whichever class $a$ or $b$ contains more terms. This was done to allow efficient joining and splitting in the backtracking search driven by the SMT solver. In our implementation the representative of a class $a$ was always a constant if $a$ contained a constant. The implicit inequality between constants was then implemented by a check that the respective equivalence classes were not both represented by a constant term. This approach fitted naturally into the egraph algorithm and explanation generation. In the experiments we observed no noticeable slowdown compared to the original approach.

**Values.** Alg. 6 requires concrete values from the UFP theory to construct a counterexample candidate. In general, the values for UFP were obtained by assigning a running number for each equivalence class that the egraph algorithm

maintained. However, there were two special cases for the values. First, if the equivalence class contained a constant, the value was that of the constant. Second, a pre-processing step in the SMT solver removed terms that only appeared on clauses that were true by construction. Since these terms could have any value, we indicated this with a special flag.

**Commutativity.** The commutativity of the functions $Co = \{ \texttt{+}, \texttt{*}_u, \texttt{*}_s, \texttt{\&}, \texttt{|} \}$ was implemented by conjoining the set $\{\circ(a,b) \leftrightarrow \circ(b,a) \mid \circ \in Co, \circ(a,b) \text{ in } P\}$ to the instance $[P]^u$ being solved. A similar approach was followed, for instance, in [CGI+17a].

### The solver for BVP

The BVP theory is solved through propositional flattening [KS08]. The solver supports the operations listed in Table 7.1, and allows the use of arbitrary bit-widths.[2] Based on an extensive testing the implementation is robust, but still prototypical in the sense that we implement no sophisticated pre-processing techniques that are available in many other bit-vector solvers (see, e.g., [BCF+07]).

Unlike many other SMT solvers (see, e.g., [HBJ+14]), we do not implement the bit-vector solver as a separate SAT solver working on the flattening and driven by the main SAT solver. Instead, we flatten the problem directly to the main SAT solver. This has several advantages: we avoid the overhead of duplicate solver instantiation, and we enable the solver to potentially learn much more intricate relationships between the flattened formula and the formula in UFP. However, an in-depth analysis of the implications of this design is beyond the scope of this thesis.

[2]The shift operations $\ll, \gg_a, \gg_l$ assume a bit-width that is a power of two.

### Theory refinement in model checking

We integrated Alg. 6 into the bounded model checker HiFrog for C programs. HiFrog obtains first the symbolic encoding of the program $P$ and a safety property $t$ through a sequence of pre-processing steps, builds then the UFP formula, and finally gradually transforms parts of the UFP formula into BVP based on truth assignments until the safety is determined. We follow the approach where safety properties are expressed as assertions in the C code. The architecture is depicted in Fig. 7.3. HiFrog maintains two SMT solvers during the execution and which are represented by the *checkSAT* calls in Alg. 6: the *main solver* for checking the satisfiability query constructed at line 5 (shown on the bottom of Fig. 7.3) and the *refinement solver* for checking the spuriousness of each counterexample at line 10 (shown on the right of Fig. 7.3). This choice was taken so that the expensive calls on the main solver would not be slowed down by unnecessary clauses at the refinement solver.

The counterexamples were flattened to propositional logic through the call to `getValues` by mapping the values in UFP to a unique bit-vector constant of the given bit width $bw$. At this stage of the development we ignored the case where the UFP solver gave more equivalence classes than what was representable in $bw$ bits, since this limitation did not affect our results.

The binding formula (see Def. 2) is updated whenever a statement $x = t$ is refined. This was done by first constructing the BVP formulas $[x]^b$ and $[t]^b$, and then adding the missing equalities to $F_B$ with the call to `computeBinding`.

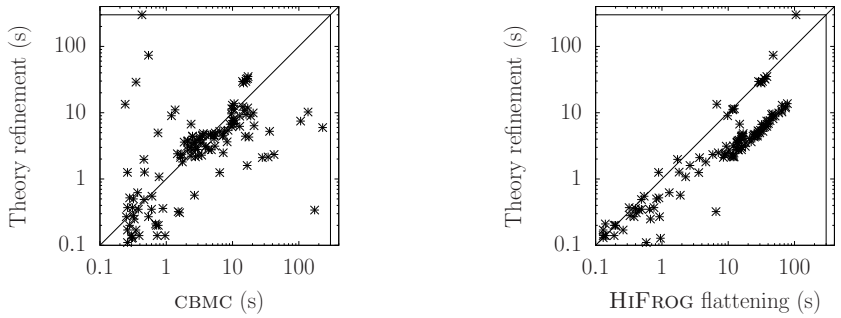


**Figure 7.4:** Timings of CBMC (*left*) and HiFrog's flattening (*right*) against HiFrog's theory refinement for the safe instances.

### 7.4.2 Experimental results

We evaluated the theory-refinement mode of HiFrog on C programs mostly coming from the software model checking competition (SV-COMP). The benchmarks were split into the *safe* (128 instances) and *unsafe* (30 instances) sets, indicating whether the bad behaviour is reachable or not. Among safe instances, 17 require refinements.

For benchmarking we used Ubuntu 14.04 Linux system with two Intel Xeon E5620 CPUs clocked at 2.40GHz and 12 Gigabyte memory limit per process using a time-out of 300 seconds CPU time. The model checker was compiled with the GNU C++ compiler and the O3 optimization level. The complete experimental results, the source code, and a virtual machine are all available at [HiF17a].

Fig. 7.4 shows the verification results on safe properties. We compared (Fig. 7.4, *left*) the HiFrog's theory-refinement mode against CBMC version 5.7, the winner of the software model checking competition falsification track in 2017.[3] In 101 cases, HiFrog was either as fast or faster than CBMC, sometimes by orders of magnitude. Furthermore, HiFrog's theory refinement mode is com-

[3]OpenSMT22: `https://scm.ti-edu.ch/repogit/opensmt2.git`, git ID: 99c960e4c; HiFrog (including CBMC that shares the CPROVER framework [cpr19] with HiFrog): `https://scm.ti-edu.ch/repogit/hifrog`, git ID b35956f2c.

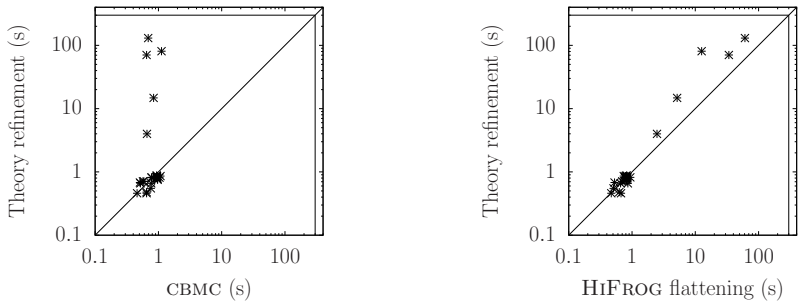


**Figure 7.5:** Timings of CBMC (*left*) and HiFrog's flattening (*right*) against HiFrog's theory refinement for the unsafe instances.

**Table 7.2:** Comparison of the heuristics against the *Min* heuristic on instances requiring refinement.

| | H0 | H1 | H2 | H3 | H4 | H5 | H6 | H7 | *Min* |
|---|---|---|---|---|---|---|---|---|---|
| #solved | **17** | 16 | **17** | **17** | **17** | **17** | **17** | **17** | 17 |
| #ref | 660 | 2218 | 1250 | 1250 | **533** | 2266 | 1442 | 1831 | 162 |
| time (s) | 538 | 223 | 257 | 317 | **123** | 166 | 147 | 158 | 46.2 |

pared against HiFrog's propositional flattening (Fig. 7.4, *right*), hence ensuring that the only difference in the solvers is in how the symbolic encoding is presented to the SMT solver. In 115 cases, the theory refinement was either as fast or faster than flattening in determining safety, providing a more convincing evidence that the theory refinement approach works well in practice.

The verification results of unsafe benchmarks are shown in Fig. 7.5. In five cases, bug detection by HiFrog was slower than the one by CBMC since HiFrog required iterative refining of all the expressions to confirm the validity of the counterexample. However, in the remaining cases, HiFrog was comparable to CBMC.

**Experiments on refinement heuristic**

Alg. 6 does not address which exact statement should be refined based on a counterexample on Line 11 in case there are several possibilities. However this selection affects the run time of the model checking and is therefore of practical interest. We consider the following three features while building a refinement heuristic:

- Traversal order: the algorithm can proceed either by choosing from $P$ the first statement (*forward order*) or the last statement (*backward order*) satisfying the condition on Line 11.
- All statements falsified by the counterexample are refined simultaneously (*simultaneous refinement*).
- All statements that depend on refined statements are refined simultaneously (*dependency refinement*).

The heuristics are as follows: H0 – Forward order; H1 – Backward order; H2 – Forward order with simultaneous refinement; H3 – Backward order with simultaneous refinement; H4 – Forward order with dependency refinement; H5 – Backward order with dependency refinement; H6 – Forward order with simultaneous and dependency refinement; and H7 – Backward order with simultaneous and dependency refinement. Based on the experimentation, the fastest solver on average resulted from using Forward order with dependency refinement. This was the heuristic we used in the results on Figs. 7.4-7.5. We briefly report on the results of the heuristics in Table 7.2 over the 17 instances of our total benchmark set where statements were refined. This benchmark set contained three crafted instances and the rest from the *bit-vector* category of SV-COMP. The row labelled

*#solved* reports how many instances the heuristic could solve before the time-out, *#ref* reports how many statements in total had to be refined over the set, and *time* reports the total run time. As a reference, the table also reports results on the heuristic *Min* that requires no run time and computes a minimum set of refinements required to prove the property.

Finally, in Fig. 7.6 we show the reduction in the number of refined statements when using the *Min* heuristic on the 17 instances. As expected, the performance of the heuristic depended on the instance, but when effective, dramatically reduced the amount of flattened statements.

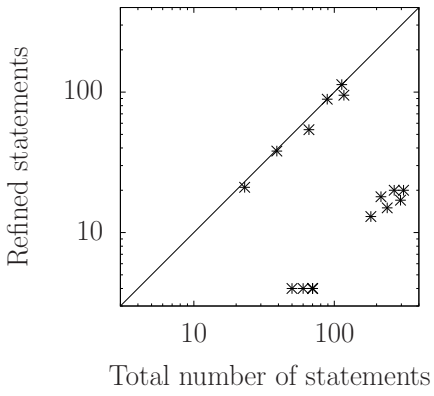


**Figure 7.6:** Number of refined statements using the *Min* heuristic with respect to total number of statements.

While the results are still preliminary mostly due to the prototype nature of the tools we are developing, we believe that they make a very strong point for the potential of the theory refinement approach in software model checking.

## 7.5 Related Work

Solving bit-vector problems with layers of theory solvers was introduced in [BCF+07] and further developed in [HBJ+14]. While we worked directly on software verification instead of bit-vectors, our approach is related, as we also used a hierarchy of solvers combined with rewriting techniques. However, we worked explicitly on the modelling language by automatically adjusting the precision to be different in different parts of the problem and adding additional constraints

that seam these parts together. In [BB09], a CEGAR-based approach was used for solving problems involving arrays by transforming an abstract representation into clauses. We differ from this approach in that we integrated the system on the theory solver level, employing in the experiments the congruence closure algorithm together with a propositional solver. To the best of our knowledge, no existing approach used this level of granularity in the modelling. Furthermore, we used counterexamples that were checked against the bit-precise implementation, and this way avoided the refinement of program parts likely needed to be refined in approaches based on layered theory solvers.

Exploiting simultaneously several theories for one verification goal is not new. For example, [GBM14] presented a system for synthesizing safe bit-precise inductive invariants for software. Compared to our work in this chapter, the refinement direction is inverted: the software was first flattened, and in case of a time-out, converted to a domain-specific theory. Furthermore, we integrated seamlessly the theories UFP and BVP into an SMT solver, whereas [GBM14] considered real arithmetics.

Uninterpreted functions have been used together with the bit-precise encoding for verifying the equivalence of Verilog designs in [BBS11, HCR+16]. The approach used machine learning to identify sub-components that can likely be abstracted. In contrast, our emphasis is on software verification and integration to the SMT solver. A related approach [KIY16] constructed test cases for scientific software by computing difference constraints from non-linear mathematical functions. This approach can be viewed as a special case of the framework we present in this chapter; the formulas we derived could also be used for generating test cases, although this is not the focus of this chapter. Similarly, incremental reduction approach [CGI+17a] combined linear real arithmetic and equality of

uninterpreted functions (EUF) for the SMT encoding of the program. The algorithm initially used EUF to abstract non-linear operators, and then used the monotonicity and the multiplication checks to identify spurious counterexample, thus avoiding simulation and code execution. Both checks might result in a refinement formula, which was added then to the current SMT encoding. Unlike ours, their approach cannot be applied as such for bit-precise reasoning.

The incremental reduction approach was extended to incremental linearisation approach for solving efficiently non-linear arithmetic problems in general in [CGI+17b, CGI+18a, CGI+18b, IGCS19]. The approach has been adopted for solving transcendental functions in MathSAT5 [CGI+17b, CGI+18b] and for solving symbolic bit-width bit-vectors problems in CVC4 [RTJB17, BT19]. The CEGAR-based incremental linearisation required the definition of the relative error approximation per function it refined (to be able to increase precision during refinement), while our work used a lighter approach and only required the bit-width configuration as an input. In Chap. 4, we reported very positive results on using the combination of EUF, LRA, and propositional flattening for encoding model checking problems. In this chapter, the work explored the possibilities in much more depth and rigour was motivated by this early result in Chap. 4.

Another program-based refinement approach was proposed in [KBH15], where the compositional program was approximated with a program-specific theory of transition systems. Our approach is orthogonal to this, as we were able to handle programs in a more general way through the eventual flattening, while the theory of transition systems could likely be integrated as an additional theory.

## 7.6 Conclusions and Future Work

We presented a new approach for abstraction refinement in software verification with SMT solvers. Our approach introduces iterative *theory refinement* and supports solving of formulas of combined theories in the SMT solver, where the binding to the theory is maintained by a series of identities in the original formula. Our main contribution is the gradual encoding process that uses the most precise theory only for a subset of all program statements, while handling the rest of the statements by using the less precise theories. This subset of the statements could either be identified by checking spurious counterexamples or simply specified by the user. Our framework can be extended by sets of theories with a partial order of refinement defined among them. In this chapter, we demonstrated the framework on the UFP theory with the partial refinement to the BVP theory. We implemented this framework in the OpenSMT2 SMT solver [HMAS16] solver and the model checker HiFrog (Chap. 4, [AAC+17]).

We study different refinement strategies and compare them against a strategy computed off-line, as well as with the encoding into propositional logic, known as flattening or bit-blasting. Improvement is seen both in the running time and in the size of the resulting formula, demonstrating that the spurious counterexamples are usually eliminated by refining a small number of statements in the formula.

In future we plan to progress in several directions. We will study theory refinement with arithmetic theories and arrays, defining a partial order among theories based on the level of abstraction/refinement that they provide. We will further improve the automatic refinement based on an analysis of the counterexamples using approaches such as interpolation. We also plan to develop more sophisticated heuristics and strategies for refinement.

# Chapter 8

# Function summarization Modulo Theories

SMT-based software verification can achieve high precision using bit-precise models or combinations between different theories. Often such approaches suffer from problems related to scalability due to the complexity of the underlying decision procedures. Achieving better performance (at the expense of precision) is done by increasing the level of abstraction of the model. However, as the level of abstraction increases, missing important details of the program model becomes problematic.

In this chapter, we handle this trade-off problem with an incremental verification approach that controls the precision of the program modules on demand. The idea is to model a program using the lightest possible (i.e., less expensive) theories that suffice to verify the desired property. We employ safe over-approximations for the program based on both function summaries and light-weight SMT theories. During the verification, once the precision is too low to prove the correctness

of a property, our approach lazily strengthens either all affected summaries or the theory through an iterative refinement procedure.

The resulting summarization framework provides a natural and lightweight approach for carrying information between different theories. An experimental evaluation with a bounded model checker for C on a wide range of benchmarks demonstrates that our approach scales well, often effortlessly solving instances where the state-of-the-art model checker CBMC runs out of resources.

## 8.1 Introduction

The specifications of software systems with *multiple properties* can be expensive to check due to a significant amount of repetitive work. We suggest overcoming this matter by operating incrementally so that the verifier can reuse results obtained during verification of different properties to avoid wasting resources. Assuming that a specification usually involves a certain amount of closely related properties, this incremental approach avoids verifying each property from scratch, and instead automatically identify and focus on small "deltas" in the verification conditions.

Verification approaches based on Satisfiability Modulo Theories (SMT) represent a program together with a specification in first-order logic. Often the specification is naturally expressible as a set of individual properties. Each of the properties in the specifications of the software can require a different encoding that is precise enough to prove the absence of spurious counterexamples of this property. In a real sense, this means that each formula requires its own *theory*. For example, some properties might be provable with a lightweight and an inexpensive encoding, such as the theory of equality and uninterpreted functions (EUF), while other properties might require expensive bit-precise reasoning. Identifying

automatically which theory is suitable for verifying each property is challenging. In the incremental verification setting maintaining such a framework gives new challenges. In this chapter, we solve this problem by designing the *theory interface* that enables migrating information among formulas in different theories. An over-approximating *function summary* [SFS12b, AAC+17] is a well-known concept in Bounded Model Checking [BCCZ99] that enables reuse of information among verification runs. Summaries are extracted using Craig interpolation [Cra57] after a successful verification run for one property and used as a light-weight replacement of the precise encoding of the corresponding functions while verifying other properties. In this chapter, we propose an algorithm that effectively *incorporates different theories* for incremental verification of multiple properties via creation, reuse, and refinement of function summaries.

To the best of our knowledge, this is the first integrated system for SMT-based incremental model checking in which a sequence of safety properties is verified. Our algorithm works as follows. Given a program, a sequence of properties to verify and an initially empty set of function summaries in several available theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$, the algorithm encodes the program and the current property using the least precise theory $\mathcal{T}_1$ and the least precise summaries available. In case the algorithm finds a proof, the result is sound since we guarantee that both the theories and the summaries always over-approximate the concrete program. Our algorithm starts with imprecise encodings since, if sufficient for proving a property, it lowers the cost of summarization and results in more compact summaries. If no proof is found, the algorithm increases the precision lazily. Assume that the problem is currently encoded using the theory $\mathcal{T}_i$. In the phase called *local refinement*, the algorithm sequentially adds summaries translated from theories $\mathcal{T}_j$ to $\mathcal{T}_i$ ($j \neq i$) and checks if the property in this encoding is provable. The algorithm enters the

second phase, *global refinement*, where the problem is encoded in a more precise theory $\mathcal{T}_{i+1}$, only when all summaries are already tried on theory $\mathcal{T}_i$. Then the algorithm returns to the local refinement again. Similarly to [SFS12b, AAC+17], our algorithm is capable of generating new function summaries and identifying actual bugs. Our refinement is driven by a counterexample-guided analysis that distinguishes spurious counterexamples from the real ones.

We have implemented the algorithm on top of the function-summarization-based bounded model checker HiFrog using the OpenSMT2 SMT solver for both solving and interpolation [HMAS16]. Our implementation supports the theories of equality and uninterpreted functions (EUF), linear real arithmetic (LRA), and bit-vectors (BV). We provide an extensive evaluation on a range of large-scale benchmarks taken from SV-COMP[1] and crafted by ourselves. The tool exhibits a competitive performance compared to the state-of-the-art.

To sum up, our contributions are as follows:

- A novel approach to incremental verification that lazily identifies, among several suitable candidates, the lightest level of encoding for each given property.
- A theory interface for exchanging function summaries among formulas in different theories.
- An algorithm to leverage both function summaries and the overall precision of the program encoding that in practice demonstrates a competitive performance on a range of large-scale programs.

The rest of the chapter is structured as follows. We motivate the work presented in this chapter with an example in Section 8.2. We provide a background

[1]Software Verification Competition, `http://sv-comp.sosy-lab.org/`, Linux Device Drivers (ldv) category.

on function summarization and SMT in Section 8.3 and describes formally how the summary conversion is carried out between different SMT encodings in Section 8.4. We present our main algorithm for combining summary refinement and theory refinement in Section 8.5. Finally, we report the experimental evaluation in Section 8.6, give a brief overview of related work in Section 8.7, and concludes this chapter in Section 8.8.

## 8.2 Motivating Example

Fig. 8.1 shows a C program with a function call containing non-linear operations and two user-defined assertions. Our approach verifies the two assertions in the code incrementally. It is not hard to see that the program is safe with respect to both assertions. However, verification of this program using bit-precise encoding is expensive.

Our algorithm tries the less precise but easier to solve theory of equality and uninterpreted functions (EUF) as the level of abstraction first, leading to successful verification of the first assertion almost immediately. The algorithm then generates and stores a summary for function `func`. To verify the second assertion, reasoning over linear real arithmetic (LRA) is necessary. Our algorithm presented later in this chapter enables to translate the summary for `func` from EUF to LRA and to reuse it to successfully verify the second assertion.

## 8.3 Background and Previous Work

Our discussion relies heavily on concepts used in SMT solving. In Sec. 2.3.1, we define the notation that we use in the presentation in this chapter.

```
1
2  int a, b, c;
3
4  void func() {
5    c = b;
6    if (a > 0) a = b;
7    int m = 0;
8    for (int i = 0; i < 100; i++)
9      m += a*b;
10   b = m;
11 }
12
13 int main() {
14   a = nondet(); b = nondet();
15   if (a <= 0) return -1;
16   func();
17   assert(a == c);
18   if (a > 0) {
19     func();
20     if (c > 10) assert(a > 7);
21   }
22   return 0;
23 }
24
```

**Figure 8.1:** Program in C with non-linear arithmetic.

### 8.3.1 Programs and summaries

In this thesis, we reduce a Bounded Model Checking (BMC) [BCCZ99] task to an SMT task. That is, a program is encoded to a quantifier-free first-order formula in a given theory $\mathcal{T}$, which is then solved for satisfiability. Our intent is to allow function calls in the considered programs and to over-approximate them by summaries whenever applicable. If the program encoding is inconsistent with the negation of safety specification, then the program is safe. For the definition of a loop-free program with summaries as a quantifier-free first-order formula see Sec 2.5.1 in Chap. 2.

Given an unsatisfiable formulas, we construct summaries using Craig Interpo-

```
// encoding of the first call of function
"func":
```

$$c_1 = b_0 \wedge a_1 = ite(a_0 > 0, b_0, a_0) \wedge m_0 = 0 \wedge \text{L\_UNW}_1 \wedge b_1 = m_{10}$$

```
// encoding of the second call of function
"func":
```

$$c_2 = b_1 \wedge a_2 = ite(a_1 > 0, b_1, a_1) \wedge m_{11} = 0 \wedge \text{L\_UNW}_2 \wedge b_2 = m_{21}$$

```
// encoding of function "main":
```

$$a_0 > 0 \wedge a_1 > 0 \wedge c_2 > 10$$

```
// encoding of the negation of the first
assertion:
```

$$\neg(a_1 = c_1)$$

```
// encoding of the negation of the second
assertion:
```

$$\neg(a_2 > 7)$$

**Figure 8.2:** Modular encoding of program from Fig. 8.1 to an SMT formula.

lation [Cra57], a widely used technique to create over-approximations in Model Checking; see Sec. 2.3.2 and Sec. 2.5.1 in Chap. 2 for formal definitions with basic examples of Craig interpolation and function summaries.

In our context, unsatisfiable formulas originate from bug-free programs, and thus the summaries express that no trace allowed by the function body leads to a violation of the considered safety specification. In order to construct and use function summaries in the context of BMC, we assume that a BMC formula is a conjunction of encodings of individual function calls. Thus, the problem of determining whether the program is safe with respect to a safety assertion $Q$ reduces to the problem of determining the satisfiability of the SMT formula

$$\bigwedge_{\hat{f} \in \hat{F}} \text{ENCODE}(\hat{f}) \wedge \neg\text{ENCODE}(Q) \implies \bot.$$

We illustrate the encoding on the following example.

**Example 9.** Fig. 8.2 shows a (simplified) encoding of program from Fig. 8.1 to an SMT formula. The formula consists of five parts: a conjunct representing function ***main***, two equivalent (modulo renaming) conjuncts representing calls of ***func***, and two conjuncts representing the negated assertions. As customary in BMC, each program variable has its indexed copies (induced by the single static assignment form). The formulas L_UNW$_i$, $i \in \{1, 2\}$, represent a loop unwinding. Note that the encoding of ***main*** consists only of the path condition to assertions, but in general it should explicitly encode all possible paths through the function body.

In [SFS12b], we presented a method to extract summaries for every function call $\hat{f}$ exploiting the proof of unsatisfiability of this formula. In a nutshell, the approach considers a conjunction of the encoding of all nested function calls from $\hat{f}$, i.e., $f_{precise} \stackrel{\text{def}}{=} \text{ENCODE}(\hat{f}) \wedge \bigwedge_{\hat{g} \in nested\_calls(\hat{f})} \text{ENCODE}(\hat{g})$, treats it as $A$, treats the rest of the program encoding (including the negation of assertion) as $B$, and interpolates. Note that the resulting interpolant $f_{sum}$ can now be used in place of $f_{precise}$ when creating the formula again because by construction $f_{precise} \implies f_{sum}$.

**Example 10.** A possible function summary for ***func*** obtained after verifying the first assertion is $(a_0 > 0) \implies (a_1 = c_1)$. It can successfully replace both calls to ***func*** (after the variables are renamed to match the second call) while verifying the second assertion.

Note that for the examples above, using two SMT precisions are enough: EUF for the first assertion, and LRA for the second one.

## 8.4 Theory-Based Model Refinement

This section presents a general framework that allows a translation back and forth among theories of SMT with different level of precision.

Our work views the problem of bounded model checking of C programs as a decision problem which is (i) decidable, and (ii) not based on Nelson-Oppen theory combination [NO79]. We may therefore concentrate in our framework on four theories of interest: the quantifier-free theories of equality and uninterpreted functions (EUF), linear real arithmetic (LRA), non-linear real arithmetic (NRA), and bit-vectors (BV)[2]. As a result, we obtain a decision procedure that has a relatively low complexity. Our framework, called *theory interface*, provides a common place from which the theories are instantiated, and to which they can also be converted back. This theory interface is not aimed to be passed to an SMT solver, but instead provides an infrastructure through which an instance from one theory can be converted to an instance from another theory.

The transformation from a theory $\mathcal{T}$ to the theory interface and back can be expressed in the theory-specific instantiations of the following rules, where $[\phi]^{\mathcal{T}}$ denotes that the expression $\phi$ is encoded using theory $\mathcal{T}$:

$$\frac{[f(\mathbf{t})]^{\mathcal{T}}}{f([\mathbf{t}]^{\mathcal{T}})} \quad \text{if } f([\mathbf{t}]^{\mathcal{T}}) \text{ in } \mathcal{T} \qquad\qquad \frac{[f(\mathbf{t})]^{\mathcal{T}}}{v_{f(\mathbf{t})}} \quad \text{if } f([\mathbf{t}]^{\mathcal{T}}) \text{ not in } \mathcal{T} \tag{8.1}$$

We use the notation $f(\mathbf{t})$ to abbreviate $f(t_1, \ldots, t_n)$, and $f([\mathbf{t}]^{\mathcal{T}})$ to abbreviate $f([t_1]^{\mathcal{T}}, \ldots, [t_n]^{\mathcal{T}})$. Above we write $f([\mathbf{t}]^{\mathcal{T}})$ *in* $\mathcal{T}$, if $f \in \Sigma$ and there is a derivation recursively using the rules (8.1) such that $f([\mathbf{t}]^{\mathcal{T}})$ is expressible in $\mathcal{T}$. We denote by $v_{f(\mathbf{t})}$ a variable that is unique to the expression $f(\mathbf{t})$. For example,

[2]For the signature of bit-vectors, we use a modification presented in Chap. 7, that preserves the high-level programming language structures to facilitate the proofs of over-approximation.

the expression $f(x, x)$ is not expressible in the theory of linear real arithmetic if $f$ is the multiplication operation and $x$ is a variable, and therefore the result of applying the rules (Eq. (8.1)) is $v_{x*x}$. To simplify slightly the notation, we define a bijection $\mathcal{M}$ that maps terms $f(\mathbf{t})$ to the variables $v_{f(\mathbf{t})}$. For completeness, we present the three rules for transforming non-atomic formulas

$$\frac{[\phi_1 \wedge \phi_2]^{\mathcal{T}}}{[\phi_1]^{\mathcal{T}} \wedge [\phi_2]^{\mathcal{T}}} \qquad \frac{[\phi_1 \vee \phi_2]^{\mathcal{T}}}{[\phi_1]^{\mathcal{T}} \vee [\phi_2]^{\mathcal{T}}} \qquad \frac{[\neg\phi]^{\mathcal{T}}}{\neg[\phi]^{\mathcal{T}}} \tag{8.2}$$

that are independent of a theory and thus common to all transformations.

### 8.4.1 Theory interface

A *theory interface* $\mathfrak{T}$ is a general representation of formulas that we use for transformation among theories. Fig. 8.3 outlines a communication among our four theories of interest; the horizontal arrows demonstrate the relation among these theories from the perspective of over-approximation. This relation is a part of the contribution of this work. Because this chapter aims at using from early on a light-weight theory that suffices for reasoning, over-approximation among theories is at the core of speeding up the solving procedure. In the rest of this section, we formally define a theory interface and establish a relation among theories in a sound way.

**Definition 4** (Theory interface $\mathfrak{T}$)**.** *Given a sequence of theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$ with signatures $\Sigma_1, \ldots, \Sigma_n$ respectively, a theory interface $\mathfrak{T}$ is a tuple $\langle \Sigma, \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle$ where $\Sigma \stackrel{\text{def}}{=} \Sigma_1 \cup \cdots \cup \Sigma_n$, and each $\mathcal{M}_i$ is a* bijective *mapping $\mathcal{M}_i : (\mathcal{S}_\Sigma \cup \mathcal{R}_\Sigma) \setminus (\mathcal{S}_{\Sigma_i} \cup \mathcal{R}_{\Sigma_i}) \to \mathcal{X}_i$ where $\{\mathcal{X}_i\}_{0<i\leq n}$ are pairwise disjoint sets of unique variables not used anywhere else.*

Intuitively, $\mathcal{M}_i$ replaces the formulas and terms that are not expressible in

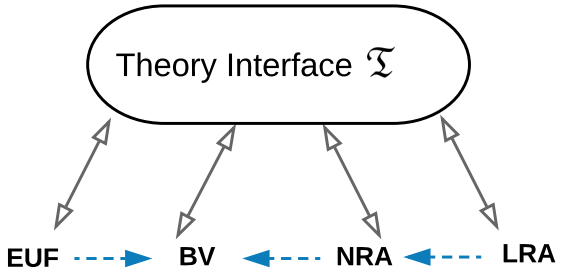


**Figure 8.3:** Theory interface between EUF, LRA, NRA, and BV.

theory $\mathcal{T}_i$ by unique fresh variables. Note that for every $\mathcal{T}_i$, $\mathcal{S}_{\Sigma_i} \subseteq \mathcal{S}_\Sigma$ and $\mathcal{R}_{\Sigma_i} \subseteq \mathcal{R}_\Sigma$.

The projection of $\mathfrak{T}$ to one of the theories $\mathcal{T}_i$ is done by the following rules. First, if $f(\mathbf{t}) \in \mathcal{S}_{\Sigma_i}$ (i.e., is expressible in theory $\mathcal{T}_i$), then it is projected to $\mathcal{T}_i$ without changes. Second, if $f(\mathbf{t}) \notin \mathcal{S}_{\Sigma_i}$ (i.e., is not expressible in theory $\mathcal{T}_i$), then we replace it by a fresh symbol $\mathcal{M}_i(f(\mathbf{t})) \stackrel{\text{def}}{=} v_{f(\mathbf{t})} \in \mathcal{X}_i$. For transformation in the opposite direction, i.e., $\mathcal{T}_i$ to $\mathfrak{T}$, we define the inverse function $\mathcal{M}_i^{-1}$ as $\mathcal{M}_i^{-1} : v_{f(\mathbf{t})} \mapsto f(\mathbf{t})$ for $v_{f(\mathbf{t})}$ in the range of $\mathcal{M}_i$.

In the following, we develop a set of translation functions to different theories and build the over-approximation relation among these translation functions. Given a formula $\phi$ in theory interface $\mathfrak{T}$ and an arbitrary theory $\mathcal{T}$, we write $Tr_{\mathcal{T}}(\phi)$ for the translation from $\phi$ to $\mathcal{T}$.

**Definition 5** (over-approximation)**.** *Let $\phi$ be a formula in $\mathfrak{T}$, and $\mathcal{T}_1$ and $\mathcal{T}_2$ two arbitrary theories. The two translation functions, $Tr_{\mathcal{T}_1}(\phi)$ and $Tr_{\mathcal{T}_2}(\phi)$ convert the original formula $\phi$ into $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. We say that $\mathcal{T}_1$ over-approximates $\mathcal{T}_2$ if no satisfying assignment exists in $\mathcal{T}_1$ for formula $Tr_{\mathcal{T}_1}(\phi)$, then it implies that there is no satisfying assignment in $\mathcal{T}_2$ for formula $Tr_{\mathcal{T}_2}(\phi)$.*

We give the specifics for the theories EUF and LRA, and provide after it also

the rules of transformation from theory interface to BV and NRA. To establish the over-approximation relation, we assume in this chapter that the programs being verified admit no overflows or underflows, and that their semantics can be exactly captured by BV.

**Definition 6** (Theory of EUF)**.** *Let $\mathcal{X}$ be a set of variables and $\mathcal{F}$ be a set of function symbols with arities. An Equality logic formula with uninterpreted functions (*EUF*) is defined by the grammar*

$$
\begin{aligned}
trm ::= & \; const \\
& \mid var \\
& \mid f(trm, \ldots, trm) \qquad \textit{where } f \textit{ is uninterpreted} \\
fla ::= & \; Bvar \\
& \mid p(trm, \ldots, trm) \qquad \textit{where } p \textit{ is uninterpreted} \\
& \mid trm = trm \mid trm \neq trm \mid \top \mid \bot \mid \neg fla \\
& \mid fla \wedge fla \mid fla \vee fla \mid
\end{aligned}
$$

*where $fla$ is a quantifier-free formula, $var \in \mathcal{X}$, $f \in \mathcal{F}$, and $const \in \mathcal{C}$. With the exception of equality and disequality ($=, \neq$), function and predicate symbols are treated as uninterpreted.*

Semantically, EUF has the axioms of reflexivity, symmetry and transitivity for the symbol of equality, and congruence axiom for function and predicate symbols $(\mathbf{x} = \mathbf{y}) \rightarrow (f(\mathbf{x}) = f(\mathbf{y}))$ and $(\mathbf{x} = \mathbf{y}) \rightarrow (p(\mathbf{x}) \leftrightarrow p(\mathbf{y}))$ where $\mathbf{x} = \mathbf{y}$ is an abbreviation for $(x_1 = y_1) \wedge \ldots \wedge (x_n = y_n)$ and $f$ and $p$ are function and predicate symbols, respectively, of arity $n$.

**Definition 7.** *A quantifier-free formula in the language of theory of Linear Real*

*Arithmetic (LRA) is defined by the following grammar:*

$$
\begin{aligned}
trm &::= const \\
&\mid var \\
&\mid const * \ var \\
&\mid f(trm, \ldots, trm) \qquad \textit{where } f \in \{+\} \\
fla &::= Bvar \\
&\mid p(trm, \ldots, trm) \qquad \textit{where } p \in \{\leq, <\} \\
&\mid \top \mid \bot \mid \neg fla \\
&\mid fla \wedge fla \mid fla \vee fla \mid
\end{aligned}
$$

*where var are variables, and const is a rational number.*

We define next a particular class of quantifier free theory of bit-vectors (BV) which is based on the definitions in Chap. 2 (Sec. 2.3.1) and Chap. 7 (Sec. 7.2). In Chap. 7, we presented theory called BVP (Bit Vectors for Programs) which was an augmented version of the theory of bit-vectors. For abbreviation we use in this chapter the BV notation. In order to be applicable in our framework, BV should comply with the restriction that no overflows are allowed.

Based on SMT-LIB2 standard the signature in BV is as follows: $\Sigma^{BV} = \{+, *, bvand, bvor, bvudiv, bvurem, bvshl, bvlshr, bvnot, bvneg\}$. We consider the predicate symbols as $\mathcal{P} = \{>, <, \leq, \geq\}$. Note that for the addition and multiplication we use the same notation i.e., "+" and "*" throughout the chapter to highlight the fact that the syntax is in common with our theory of interest. Therefore syntactically they can be used in the transformation rules.

However, the task of interpretation of each function symbol must be delegated to the corresponding theory solver.

### 8.4.2 Encoding of theory interface into specific theories

Light-weight theories help removing overly complex or irrelevant details from the encoding of a program whenever possible. We define the following rules for the theory-specific part of the transformation from $\mathfrak{T}$ to $EUF$:

$$\begin{array}{l} \dfrac{[v]^{EUF}}{v} \; v \text{ is a variable or a constant} \\ \\ \dfrac{[t_1 = t_2]^{EUF}}{[t_1]^{EUF} = [t_2]^{EUF}} \\ \\ \dfrac{[t_1 \bowtie t_2]^{EUF}}{(\neg[t_1 = t_2]^{EUF}) \wedge ([t_1]^{EUF} \bowtie [t_2]^{EUF})} \bowtie \in \{>, <\} \\ \\ \dfrac{[f(\mathbf{t})]^{EUF}}{f([\mathbf{t}]^{EUF})} \quad \text{otherwise} \end{array} \tag{8.3}$$

Note that in the third rule of (8.3), if the function symbol $>$ or $<$ is applied over the terms of theory interface, it can be simply translated into a disequality in EUF. All the other cases in the signature of theory interface which cannot be applied in the first three rules such as $\{\leq, \geq, \ldots\}$ are handled by the fourth rule.

**Theorem 6.** *For every formula $\phi \in \mathfrak{T}$, EUF over-approximates BV.*

Over-approximations between theories is defined in Def. 5.

*Proof.* We show that every model in BV can be translated to a model in EUF.

Assume that there is a satisfying assignment in BV, such that $a = b$ holds for two bit-vectors $a$ and $b$. This can be trivially translated to an equality $a = b$ in EUF.

In case of equality of two function applications $f(a) = f(b)$, we utilize the congruence rule in EUF, assuming that each function in BV is implemented as a deterministic circuit. □

We define the following rules to transform $\mathfrak{T}$ to $LRA$[3]:

$$
\begin{aligned}
&\frac{[v]^{LRA}}{v} \quad v \text{ is a variable or an integer constant} && (8.4.2)\\
&\frac{[t_1 = t_2]^{LRA}}{([t_1]^{LRA} \leq [t_2]^{LRA}) \wedge ([t_2]^{LRA} \leq [t_1]^{LRA})} && (8.4.1)\\
&\frac{[t_1 \ + \ t_2]^{LRA}}{[t_1]^{LRA} \ + \ [t_2]^{LRA}} && (8.4.3)\\
&\frac{[-t_1]^{LRA}}{(-1) * [t_1]^{LRA}} && (8.4.4)\\
&\frac{[t_1 * t_2]^{LRA}}{[t_1]^{LRA} * [t_2]^{LRA}} \quad t_1 \text{ or } t_2 \text{ is an integer constant} && (8.4.5)\\
&\frac{[f(\mathbf{t})]^{LRA}}{\mathcal{M}(f(\mathbf{t}))} \text{ otherwise} && (8.4.6)
\end{aligned}
\tag{8.4}
$$

The rule (8.4.6) uniquely associates the expression with a fresh variable. Essentially this rule is used for over-approximation of all the expressions that cannot be expressed in sufficient precision in LRA. The rules (8.4.2) and (8.4.5) operate only on integer constants in order to preserve the soundness of translation between LRA and BV. Example 11 illustrates this case in detail.

**Example 11** (**Over-approximation of** BV **by** LRA)**.** Consider the following excerpt of a program written in C: `int x = 1; int y = 0.5 * x; assert ( y == 0 );` Let $\phi \stackrel{\text{def}}{=} x = 1 \wedge 0.5 * x = 0$ represent the corresponding SMT repre-

[3]We assume that before undergoing a transformation, a preprocessing is done for the sake of normalization, e.g., $-1 * 2 * x$ is normalized to $-2 * x$.

sentation. Following the semantics of C, a bit-precise encoding of $\phi$ is satisfiable since $0.5 * 1$ is truncated to 0. However, an LRA-encoding of $\phi$ is unsatisfiable. According to Def. 5, this means that LRA does *not* over-approximate BV. In order to get that over-approximating behaviour, we impose restrictions on LRA rules (8.4.2) and (8.4.5) and apply rule (8.4.6) when these restrictions are not met. The translation applied to $\phi$ results in $x = 1 \wedge v_{0.5*x} = 0$ which is satisfiable in LRA. The same restrictions are imposed in NRA.

**Theorem 7.** *For every formula $\phi \in \mathfrak{T}$, LRA over-approximates BV.*

Over-approximations between theories is defined in Def. 5.

*Proof.* We show that every model in BV can be translated to a model in LRA. Assume that there are no overflows or underflows in BV. This guarantees that the models of all arithmetic operations in BV are also models in LRA. □

The rules for transforming from $\mathfrak{T}$ to *NRA* are as follows:

$$\frac{[v]^{NRA}}{v} \; v \text{ is a variable or a constant}$$

$$\frac{[t_1 = t_2]^{NRA}}{([t_1]^{NRA} \leq [t_2]^{NRA}) \wedge ([t_2]^{NRA} \leq [t_2]^{NRA})} \tag{8.5}$$

$$\frac{[t_1 \bowtie t_2]^{NRA}}{[t_1]^{NRA} \bowtie [t_2]^{NRA}} \bowtie \text{ is a function symbol in } NRA, e.g., \bowtie \in \{+, -, * \}$$

Regarding the transforming to *NRA*: the "otherwise" fall back,

$$[\text{otherwise}]\mathcal{M}(f(\mathbf{t}))[f(\mathbf{t})]^{NRA}$$

, is not required here as we never use it. This is because we assume that all formula and terms from *NRA*, *LRA*, *BV* and *EUF* are expressible in *NRA* (see Def. 4 and Fig. 8.3) under the encoding of the verification problem in HiFrog.

In the following the rules for translation from $\mathfrak{T}$ to $BV$ are as follows:

$$\frac{[v]^{BV}}{v} \qquad v \text{ is a variable or a constant}$$

$$\frac{[t_1 = t_2]^{BV}}{[t_1]^{BV} = [t_2]^{BV}}$$

$$\frac{[t_1 \bowtie t_2]^{BV}}{[t_1]^{BV} \bowtie [t_2]^{BV}} \qquad \bowtie \text{ is a predicate symbol or binary function symbol in } BV,$$
$$i.e., \bowtie \in \{bvand, bvor, +, *, bvudiv, bvurem, bvshl, bvlshr\}$$

$$\frac{[\triangle t_1]^{BV}}{\triangle [t_1]^{BV}} \qquad \triangle \text{ is a unary function symbol in } BV,\ i.e., \triangle \in \{bvnot, bvneg\}$$

$$\frac{[f(\mathbf{t})]^{BV}}{\mathcal{M}(f(\mathbf{t}))} \text{ otherwise}$$

(8.6)

### 8.4.3 Decoding theories to the theory interface

The previous section describes the instantiation from the theory interface to a specific theory of interest. This section presents the inverse, that is, transforming from a theory to the theory interface. Such steps are necessary in order to build

the over-approximation relation among theories. The key insight is to use the mapping $\mathcal{M}^{-1}$.

The transformation from *EUF* to theory interface $\mathfrak{T}$ is defined by the following rules:

$$\frac{[t_1 = t_2]^{\mathfrak{T}}}{[t_1]^{\mathfrak{T}} = [t_2]^{\mathfrak{T}}}$$

$$\frac{[v]^{\mathfrak{T}}}{v} \; v \text{ is a variable or a constant} \tag{8.7}$$

$$\frac{[f(\mathbf{t})]^{\mathfrak{T}}}{f([\mathbf{t}]^{\mathfrak{T}})}$$

Similarly, the rules for transforming from *LRA* to theory interface $\mathfrak{T}$ are as follows:

$$\frac{([t_1]^{\mathfrak{T}} \leq [t_2]^{\mathfrak{T}}) \wedge ([t_2]^{\mathfrak{T}} \leq [t_1]^{\mathfrak{T}})]}{[t_1]^{\mathfrak{T}} = [t_2]^{\mathfrak{T}}}$$

$$\frac{[t_1 \bowtie t_2]^{\mathfrak{T}}}{[t_1]^{\mathfrak{T}} \bowtie [t_2]^{\mathfrak{T}}} \text{ is a function or predicate symbol in } LRA$$

$$\frac{[v]^{\mathfrak{T}}}{v} \; v \text{ is a variable or a constant}, v \notin dom(\mathcal{M}^{-1}) \tag{8.8}$$

$$\frac{[v]^{\mathfrak{T}}}{\mathcal{M}^{-1}(v)} \; v \in dom(\mathcal{M}^{-1})$$

The rules for transforming from *NRA* to theory interface $\mathfrak{T}$ are as follows:

$$\frac{([t_1]^{\mathfrak{T}} \leq [t_2]^{\mathfrak{T}}) \wedge ([t_2]^{\mathfrak{T}} \leq [t_1]^{\mathfrak{T}})]}{[t_1]^{\mathfrak{T}} = [t_2]^{\mathfrak{T}}}$$

$$\frac{[v]^{\mathfrak{T}}}{v} \; v \text{ is a variable or a constant} \tag{8.9}$$

$$\frac{[f(\mathbf{t})]^{\mathfrak{T}}}{f([\mathbf{t}]^{\mathfrak{T}})}$$

The rules for transforming from $BV$ to theory interface $\mathfrak{T}$ are as follows:

$$\frac{[t_1 = t_2]^{\mathfrak{T}}}{[t_1]^{\mathfrak{T}} = [t_2]^{\mathfrak{T}}}$$

$$\frac{[t_1 \bowtie t_2]^{\mathfrak{T}}}{[t_1]^{\mathfrak{T}} \bowtie [t_2]^{\mathfrak{T}}} \qquad \bowtie \text{ is a binary function symbol in } BV,$$
$$e.g., \bowtie \in \{bvand, bvor, +, *, bvudiv, bvurem, bvshl, bvlshr\}$$

$$\frac{[\triangle t_1]^{\mathfrak{T}}}{\triangle [t_1]^{\mathfrak{T}}} \qquad \triangle \text{ is a unary function symbol in } BV, \triangle \in \{bvnot, bvneg\}$$

$$\frac{[v]^{\mathfrak{T}}}{v} \qquad v \text{ is a variable or a constant}, v \notin dom(\mathcal{M}^{-1})$$

$$\frac{[v]^{\mathfrak{T}}}{\mathcal{M}^{-1}(v)} \qquad v \in dom(\mathcal{M}^{-1}) \tag{8.10}$$

Determining satisfiability in an over-approximative theory does not guarantee that the formula is satisfiable in a more precise theory since the satisfiability might have been introduced by the abstraction. In such cases, the strength of the formula must be enhanced through techniques such as refinement. In the next section, we discuss how to use the theory-based model refinement idea in a model checking algorithm.

## 8.5 Summary and Theory-Aware Model Checking

Our novel approach to incremental bounded model checking is presented in Alg. 7. It takes as input a program with a sequence $\langle Q_1, \ldots, Q_m \rangle$ of safety assertions that are to be verified, and a sequence of theories $\langle \mathcal{T}_1, \ldots, \mathcal{T}_n \rangle$, such that for each $i$ and

**Algorithm 7:** VERIFY$(P, \langle \mathcal{T}_1, \ldots, \mathcal{T}_n \rangle, \langle Q_1, \ldots, Q_m \rangle)$

**Input:** Program $P$ with function calls $\hat{F}$, sequence of theories $\langle \mathcal{T}_1, \ldots, \mathcal{T}_n \rangle$; sequence of safety assertions $\langle Q_1, \ldots, Q_m \rangle$
**Output:** Verification result: {**Safe**, **Unsafe** }

```
1 for each T_j do
2 |   for each f̂ ∈ F̂ do σ_{T_j}(f̂) ← true;
3 for each Q_i do
4 |   for each T_j do
5 |   |   ⟨result, σ_{T_j}⟩ ← SumRef(P, T_j, ⟨σ_{T_1}, ..., σ_{T_n}⟩, Q_i);
6 |   |   if result = Safe then break;
7 |   |   if j = n then return Unsafe;
8 return Safe;
```

$j$, $i < j$, $\mathcal{T}_j$ is *not* an over-approximation of $\mathcal{T}_j$[4]. For simplicity, we assume that all assertions are located in the entry function (i.e., $\hat{main}$), but our implementation does not have this restriction. We refer to $\sigma_{\mathcal{T}_j}(\hat{f})$ as to a summary for function $f$ which is encoded in theory $\mathcal{T}_j$. Note that the function summary is initialized with the weakest possible summary, namely *true*. The algorithm searches for a first assertion which does not hold and then terminates with the **Unsafe** result. If no such assertion is found, the algorithm terminates with the **Safe** result.

Alg. 7 maintains a set of mappings for each function call and each theory to a summary formula that over-approximates the behaviour of the source function and is expressible in the theory. These summary formulas are initially true but are refined after a verification run of each assertion $Q_i$. Importantly, they are reused by a verification run of the next assertion $Q_{i+1}$.

An algorithm for verifying an assertion $Q$ with function summaries is shown in Alg. 8. It starts by encoding the entry function in a given theory $\mathcal{T}$ and conjoins it with the negation of encoding of $Q$ in $\mathcal{T}$. If this formula $\varphi$ is unsatisfiable, then $Q$ holds, manifesting the weakest possible summary **true** was adequate for all nested function calls from $\hat{main}$. Otherwise, our algorithm starts gradually

[4]In our implementation, we chose $\mathcal{T}_1$ = EUF, $\mathcal{T}_2$ = LRA, and $\mathcal{T}_3$ = BV.

**Algorithm 8:** SUMREF$(P, \mathcal{T}, \langle \sigma_{\mathcal{T}_1}, \ldots, \sigma_{\mathcal{T}_n} \rangle, Q)$

**Input:** Program $P = (F, f_{main})$ with function calls $\hat{F}$, theory $\mathcal{T}$; sequence $\langle \sigma_{\mathcal{T}_1}, \ldots, \sigma_{\mathcal{T}_n} \rangle$ of mappings of function calls to their summaries; $Q$: safety assertion to verify
**Output:** Verification result: {**Safe**, **Unsafe** }, updated $\sigma_{\mathcal{T}}$
**Data:** $\varphi$: BMC formula, $WL \subseteq \hat{F}$, $Pr$: precision mapping for function calls, $CEX$: counterexample

```
1  φ ← ENCODE_T(m̂ain) ∧ ¬ENCODE_T(Q);
2  for each f̂ ∈ F̂ do Pr(f̂) ← 0;
3  while true do
4      ⟨result, CEX⟩ ← SOLVE(φ);
5      if result = SAT then
6          WL ← GETCALLSWITHWEAKSUMMS(CE);
7          if WL = ∅ then return Unsafe;
8          for each f̂ ∈ WL do
9              if Pr(f̂) < n then
10                 Pr(f̂) ← Pr(f̂) + 1;
11                 ψ ← σ_{T_{Pr(f̂)}}(f̂);
12                 φ ← φ ∧ TRANSLATE_T(ψ);
13             else
14                 φ ← φ ∧ ENCODE_T(f̂);
15     else
16         for each f̂ ∈ F̂ do
17             σ_T(f̂) ← σ_T(f̂) ∧ GETITP_T(φ, f̂);
18         return ⟨Safe, σ_T⟩;
```

strengthening the formula $\varphi$ by adding summaries of the function calls responsible for the satisfiability of $\varphi$. We rely on a method described in [SFS12b] to get models of satisfiable formulas and identifying the "reason" for their satisfiability.

Our new contribution is a method to refine summaries based on lazy enumeration of available theories. In particular, Alg. 8 maintains a level of precision for each function call. In each round of refinement, if a function call $\hat{f}$ requires strengthening, its level of precision is increased by one, and a summary of that level, if available, is conjoined to $\varphi$. The key ingredient here is the set of translation rules, described in the previous section, that allow effectively reusing formulas among theories. Note that the translation process is not direct but operates via a theory interface (omitted from the pseudo-code in order to save space).

In order to prove the soundness of Alg. 7, we need to show that a summary in one theory can be reused in another theory. In other words, the correctness of Alg. 7 depends on the correctness of transferral of summaries from one theory to another theory. To this end, we connect the over-approximations via function summarization with the over-approximations via less precise theory. The following theorem captures formally the correctness of summary transformation through theory interface.

**Theorem 8.** *Let $f$ be a function, $f^{\mathcal{T}}_{sum}$ be a summary of $f$ obtained from $f^{\mathcal{T}}_{precise}$, and $f^{\mathcal{T}'}_{sum}$ be a translation of $f^{\mathcal{T}}_{sum}$ to theory $\mathcal{T}'$. Then $f^{\mathcal{T}'}_{sum}$ is also a summary of $f$.*

*Proof.* First, notice that by translating back $f^{\mathcal{T}}_{sum}$ to the theory interface using the rules in (8.1) we obtain an over-approximating representation $f_{sum}$ of $f_{precise}$. This follows from the properties of the translation. Next, by translating $f_{sum}$ to theory $\mathcal{T}'$ using rules in (8.1) we obtain an over-approximating formula $f^{\mathcal{T}'}_{sum}$ of $f_{sum}$. Finally, by transitivity $f^{\mathcal{T}'}_{sum}$ over-approximates $f_{precise}$, and hence $f^{\mathcal{T}'}_{sum}$ is a summary of $f$ as stated in the theorem. □

Note that the fact that $f^{\mathcal{T}'}_{sum}$ is a summary of $f$ is sufficient for correctness of using $f^{\mathcal{T}'}_{sum}$ instead of $f^{\mathcal{T}'}_{precise}$ in next verification tasks in case of unsatisfiability results. It is not required that $f^{\mathcal{T}'}_{sum}$ over-approximates $f^{\mathcal{T}'}_{precise}$. In case of over-approximating theory $\mathcal{T}'$ it may happen that the full encoding of a function $f$, $f^{\mathcal{T}'}_{precise}$, is not sufficient to prove a property while a summary obtained from a different theory might be enough.

## 8.6 Implementation and Evaluation

In this section we present details regarding the implementation of the additional functionality required for theory-aware summary refinement algorithm with its evaluation in HiFrog.

### 8.6.1 Implementation of the theory-aware summary refinement algorithm

We have implemented our summary and theory refinement algorithm on top of HiFrog, an SMT-based incremental bounded model checker. As a back-end, HiFrog used the OpenSMT2 SMT solver [HMAS16] which is equipped with a flexible interpolation framework for EUF [AHAS17] and LRA [AHS17] for computing function summaries. Technical information about the set-up of the tool and evaluation results are available at [Sum18].

With the reported experiments, our goal was to understand how bounded model checking can benefit from using over-approximative techniques based on function summaries obtained from SMT theories. We, therefore, compared our implementation against CBMC v5.8 [KT14], the most efficient bounded model checker based on the results of Competition on Software Verification SV-COMP [Com18]. In this chapter, we (1) automated the theory aware-refinement process (previously has been required manual intervention) and (2) applied summaries among theories (previously had no support at all). In the following, we present an explicit experimental comparison against our earlier version to highlight the usefulness of the proposed algorithm.

We instantiated the summary and theory refinement framework as described by Alg. 7 and Alg. 8 with three theories: EUF, LRA and BV (using a standard

encoding to propositional logic). In the global refinement phase of Alg. 7, the program was first encoded in EUF. In case of an unsuccessful verification with EUF, the entire program was encoded in LRA. Where the verification with LRA failed, the entire program fell back on bit-blasting. In the local refinement phase, in each of these stages, summaries of functions were used (when available) and refined on demand. After a successful verification run, summaries were extracted in the current theory and were available for verification of the subsequent assertions. Using the framework described in Sec. 8.4, these summaries were translated to different theories on-demand.

In our prototype implementation, only EUF and LRA theories exchanged summaries. However, before the precise bit-blasting of the entire program, it is possible to bit-blast the more abstract EUF and LRA summaries. While this feature is currently under development, we believe that it will lead to smaller and more compact proofs and thus improve the efficiency of the entire tool. Similarly, the inverse direction of extracting high-level information from bit-precise summaries remains a future work.

### 8.6.2 Experimental results

For benchmarking we used an Ubuntu 16.04 Linux system with two Intel Xeon E5620 CPUs clocked at 2.40GHz. We limited the memory consumption to 2 Gigabytes and the CPU time to 200 seconds per process.

We chose 109 C programs from the `ldv` category of SV-COMP [Com18] that either CBMC or HiFrog could solve within our time and memory limits. Our choice of the `ldv` benchmarks was justified because they exercised our algorithm in an interesting way due to containing many assertions and functions, and being relatively large. We excluded programs where CBMC reported an internal error.

In addition, we included 31 tricky hand-crafted smaller programs to stress-test our implementation. On average, the benchmarks had 10,000 lines of code, the longest ones reaching to 35,000 lines of code.

In total, our benchmark set contained 140 C programs and 500 assertions (verification tasks) placed inside these programs. 215 of these assertions were recognized as unreachable statements by the entry function in the C program. We excluded them from our study and focused on those tasks that require a full solving procedure. This narrowed down our set to 285 verification tasks.

In the following, we provide more details on statistics we collected after the extensive evaluation of our algorithm against CBMC and three individual verification approaches from the initial version of HiFrog (Chap. 4), namely pure EUF, LRA, BV. Table 8.1 gives statistics on our benchmark set. The column **Solved** indicates the number of benchmarks which were solved by each tool within the time and memory limits. In total HiFrog solved 24 more benchmarks than CBMC[5]. Among 98 benchmarks for which HiFrog succeeded to return an answer within the time and memory limits, 24 benchmarks were *unsafe* and 74 benchmarks were *safe*. Interestingly, the average running time for unsafe benchmarks was longer (78 s) than the one for safe ones (48 s). This can be explained by our observation that in the unsafe cases, an iterative refinement of all the summaries was required to confirm the validity of the counterexample. However, in the safe cases, HiFrog was comparable to CBMC.

As can be seen from the column **Time-outs**, CBMC performed better than HiFrog on SV-COMP benchmarks, but it failed on almost 60% of our crafted benchmarks. As can be seen from the column **Memory outs**, HiFrog solved

[5]Since many of our benchmarks include non-linear arithmetic, we also tried CBMC with the experimental `-refine` option. This did not significantly change the results, and therefore we report here the results obtained with the default options of CBMC.

**Table 8.1:** HiFrog against CBMC, and the initial version of HiFrog (Chap. 4) with respect to pure EUF, LRA, and BV solving (**#sv** is the number of benchmarks from SV-COMP, and **#craft** is the number of our tricky hand-crafted benchmarks).

| Tools | **Solved** | | **Time-outs** | | **Memory outs** | | **Unknown** | |
|---|---|---|---|---|---|---|---|---|
| | #sv | #craft | #sv | craft | #sv | #craft | #sv | #craft |
| HiFrog | **67** | **31** | 32 | **0** | **10** | 0 | - | - |
| CBMC | 63 | 11 | **28** | 20 | 18 | 0 | - | - |
| EUF only | 49 | 0 | 38 | 0 | **10** | 0 | 12 | 31 |
| LRA only | 48 | 1 | 40 | 0 | 11 | 0 | 10 | 30 |
| BV only | 43 | 4 | 33 | 4 | 33 | 23 | - | - |

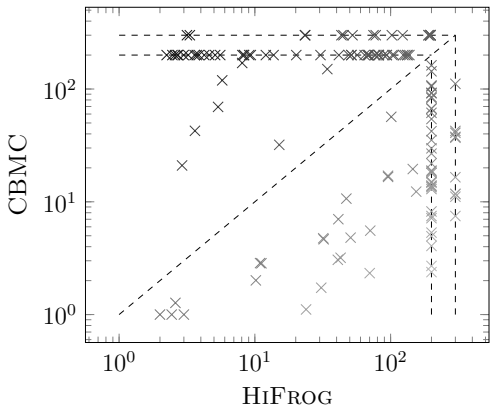


**Figure 8.4:** HiFrog vs CBMC (outer horizontal and vertical lines refer to memory limit of 2GB, and the inner lines refer to time-out at 200 s).

eight more SV-COMP benchmarks, on which CBMC immediately exceeded the memory limits. Overall, the experiments showed that HiFrog was able to solve more benchmarks, and both times out and runs out of memory were less often than CBMC.

Fig. 8.4 gives a scatter plot representing a more detailed performance comparison of HiFrog and CBMC. Each cross in the figure stands for a single benchmark with the running time of HiFrog on the x-axis, and the running time of CBMC on the y-axis. The crosses on the outer lines correspond to executions that exceeded the memory limit of 2GB, and the crosses on the inner lines correspond to

executions that exceeded the time limit of 200 s. A large amount of crosses on the top horizontal lines lets us conclude that HiFrog was able to solve benchmarks which were challenging for CBMC. Furthermore, the solving was relatively fast in these cases.

The last three rows in Table 8.1 explain how our novel algorithm in HiFrog performed compared to the initial version of HiFrog (Chap. 4), in which summary reuse was naïve and manual with respect to successive assertions. Because this functionality was not directly available in the older HiFrog, we prepared a set of helper scripts so that the older HiFrog could process assertions one after the other with possible re-use of the summaries. As expected, EUF and LRA had a large number of unsafe results, 43 and 40 respectively. We marked such results as *unknown* since due to the abstract nature of EUF and LRA the results were possibly spurious and thus cannot be trusted. By comparison, all unsafe results returned by our new algorithm correspond to actual bugs. Verifying with BV revealed that a large number of benchmarks (56 instances) exceeded the memory limit, manifesting the cost of bit-blasting, which was avoided in our new approach whenever possible.

In conclusion, we find it encouraging that the techniques described in this chapter provide such an impressive performance increase in our model checking procedure. Considering both the effectiveness and the downsides of our approach, in overall the evaluation results showed a significant positive impact on the effectiveness and efficiency of verification of large-scale and multi-property benchmarks. Although we acknowledge that these initial results obtained with the 140 instances might not be enough to draw a decisive conclusion, the results do justify future efforts into extending the benchmarking, among others, to large-scale instances with multiple user-defined assertions.

## 8.7 Related Work

Our work in this chapter was built on top of FunFrog [SFS12b], an approach for extracting and reusing interpolation-based function summaries in the context of Bounded Model Checking. The original work in FunFrog focused only on propositional logic and did not consider the rich field of first-order theories available in modern SMT solvers. Hence, despite behaving incrementally, FunFrog was expensive in many cases in practice.

In Chap. 4, we generalised the use function summaries by translating them to various SMT theories. However, Chap. 4 offered a support of different levels of abstraction, but the information obtained from one level of abstraction could only be reused at the same level of abstraction. Our new approach in this chapter has no such limitation; it converted and used information in the form of function summaries obtained from the current level of encoding when working on different levels of encoding.

The idea of using an abstract description of the bit-precise level of encoding has been applied successfully in hardware designs [ALS08] and software [HAE+17, HR17] verification. The approaches used a different level of encoding for different parts of the problem. These approaches typically started with uninterpreted functions and gradually refined to bit-level precision to rule out spurious counterexamples when necessary, while mixing different levels of encoding to verify a single property. Unlike these approaches, we did not mix different levels of encoding but shifted to more precise encoding globally, when the previous level of abstraction was insufficient. A single level of encoding allowed us to extract useful information in the form of function summaries from successful verification runs and to reuse that information in the next verification run.

Both interpolants and function summaries have been heavily used in model checking techniques. Interpolants have been commonly used as a means of abstraction. Since McMillan's first application of interpolants in formal verification [McM03], interpolation has been applied in algorithms with various extensions in model checking [CMNQ06, McM06, VG09, HHP10, ABG+12, AM13, RHK13, CPP14, McM14, VGM15, FSS17, FB18, IX18, IX19]; the model checkers CPAchecker [BK11], SeaHorn [GKKN15], Ultimate Automizer [HCD+18] and others, leveraged interpolants in some form.

Function summaries date back to Hoare logic [Hoa71] where a pair of pre-condition and post-condition can be seen as an over-approximating function summary. Besides computation of function summaries using interpolation, function summaries have been computed using data-flow analysis [RHS95, BR00, BKW07] and iterative discovery of modification of variable values, used in model checkers Saturn [XA05] and Calysto [BH08]. However, all of these applications are orthogonal to our approach in incremental model checking.

## 8.8 Conclusion and Future Work

We have presented a novel SMT-based approach to incremental verification scalable to large-scale programs with multiple properties. Our key idea is to exploit both the function summaries and the overall precision of the program encoding lazily. That is, among several theories available for the encoding, we have proposed to identify the lightest one suitable for each given property. To exploit laziness, we have designed a theory interface which enables the exchange of function summaries among formulas in different theories and avoids an expensive theory combination. Thus, our proposed algorithm performs both *local refinement* and *global refine-*

*ment* on demand. We were able to prove the effectiveness of our algorithm in practice, by implementing the approach on top of the HiFrog tool and carrying out an extensive experimental evaluation on the SV-COMP [Com18] benchmarks. Our results show that in comparison to a state-of-the-art model checker CBMC, our tool can solve more instances within the same limits on time and memory.

**Future work.** In the future, we intend to study the applicability of this approach to other areas of program verification, such as upgrade checking [FSS17], which considers a task of verification of somewhat related programs against the same property (as opposed to verification of the same program against somewhat related properties, as in the context of this chapter). We also plan to apply the developed ideas in algorithms such as software verification based on IC3 [Bra11], where correctness of unbounded programs is reduced to finding general proofs for a sequence of verification conditions that is generated on-the-fly.

# Chapter 9

# Conclusions and Future Work

The model checking technique is one of the common and successful approaches for verification of systems. Different classes of methods of model checking have been developed to deal better with the state explosion problem, as well with the computability problem in software since the development of the original technique in the 1980s. In this thesis, I have investigated modelling software systems with abstractions via approximative models with a set of complementary techniques for abstraction refinement once a model could not capture software behaviour relevant to the correctness in the current level of abstraction. I have considered two different classes of modelling via approximation; these were, the bounded model checking technique (for under-approximative models) and the satisfiability modulo theories (SMT) reasoning framework (for over-approximative models). In addition to these, I have explored the usage of the counterexample-guided-abstraction-refinement (CEGAR) framework for different verification tasks to be able to capture missing behaviour relevant to the correctness in different level of abstractions.

In this thesis, we have presented a new framework for SMT-based incremental

bounded model checking with function summaries and novel CEGAR-based techniques to handle the problems above for different verification tasks. The novelty of this work has been in the unique combination of SMT-based model checking techniques with additional abstraction and abstraction refinement techniques for efficient software verification. These techniques included bounded model checking, incremental verification and function summaries, theory refinement, the lattice-based counterexample abstraction refinement (LB-CEGAR-BMC and LB-CEGAR) and theory-aware summary refinement. We have described the implementation of the framework in a new model checker HiFrog (in Chap. 4), with the additional techniques (in Chap. 4-Chap. 8).

Through HiFrog, we have explored the ability of our techniques to find a model expressive enough to capture software behaviour relevant to the correctness yet being sufficiently high-level to prevent reasoning from becoming prohibitively expensive for different verification tasks (e.g., software with many properties, software with library function calls, and software with bit-vectors operators). We have analysed and examined the effectiveness and the downsides of the refinement approaches. Overall, through the evaluation, we demonstrated a significant positive impact on the effectiveness and efficiency of the verification of complex benchmarks. Furthermore, we investigated the negative initial results obtained with the benchmarks and believe the current results justify future efforts into developing and extending each of the refinement techniques.

To conclude, the SMT-based model checking approach is capable of verifying complex and larger software dealing efficiently with safe and unsafe instances by modelling the verification problem in different SMT logics while incorporating the smart and novel refinement techniques into the bounded model checking process.

I summarise the contributions of the thesis by presenting our methodology

details in Sec. 9.1. In Sec. 9.2, I describe the future work direction for the SMT-based incremental BMC approach and each of the refinement techniques.

## 9.1 Summary of the Thesis

We have presented and described a novel SMT-based incremental model checking framework with several different abstraction refinement techniques. Each of the refinement techniques dealt with different difficulty that resulted from an approximate model. We have generalised a single refinement flow for all four abstraction refinement techniques in the thesis in our SMT-based incremental model checking framework with function summaries.

We have described the implementation of HiFrog with its architecture and a rich set of features. The set of features included (i) a support to different encoding precisions with SMT logic, (ii) the use of function summaries and user-defined summaries, (iii) obtaining summaries through interpolation with several interpolation algorithms (depends on the theory, and other input parameters), (iv) summary compression, and (v) other basic features. Besides, to handle the over-approximation nature of SMT function summaries, we have introduced a summary refinement algorithm. The counterexample-guided summary refinement algorithm identified and marked summaries directly involved in an error detected due to a spurious counterexample. In the next refinement iteration, these summaries were replaced by the precise function representations. In addition, we have described an optimization for traversing reachability properties by constructing an assertion implication relation between close asserts in code to reduce the verification time.

In the evaluation of the basic implementation of HiFrog, we observed that model checking with the EUF and LRA-based summarisation was extremely ef-

ficient in comparison to propositional logic, yet, the light-weight theories had a high rate of false unsafe results. We have identified two reasons for this issue: (i) the over-approximative nature of the light-weight theories, (ii) lack of support of operations at the modelling stage when using SMT logics (e.g., shift-left Boolean operation or modulo operation on integers). We have proposed to decrease the false results rate with additional abstraction refinement techniques.

We examined the model checking process of programs with library functions which their correctness depends on the output of a library function. We have noticed that once the description of these functions is missing in light-weight theories, the model checking process treated these functions as uninterpreted functions. We observed that modelling these programs with light-weight theories has led to false results (that is, reporting a safe program is unsafe due to over-approximations).

For the refinement of programs with a library function, we have suggested using properties of the mathematical function in the context of bounded model checking. We have partially ordered the equations and inequalities in a subset lattice of guarded literals by inclusion relation via a new lattice construction algorithm. We have formalised a refinement process, the LB-CEGAR-BMC algorithm, in the context of bounded model checking with these lattices. The refinement process depends on a lattice traversal algorithm. The original lattice traversal algorithm we have proposed in Chap. 5 was in the context of bounded model checking. The refinement algorithm has used an initial high-level description of the function for different sub-domains of the input. Following the order of statements with an occurrence of a library function with respect to the location of the entry point and an assert statement in the loop-free program, the lattice traversal algorithm has traversed to an upper subset of guarded literals till modelling with the function's full definition based on the lattice structure and spurious counterexamples.

In the evaluation of LB-CEGAR-BMC on benchmarks with modulo operation, we observed that LB-CEGAR-BMC consumed fewer resources with lattice than with UDS, especially as the number of summaries loaded to HiFrog had grown, and had (with LRA) the best time consumption results. However, LB-CEGAR-BMC failed to prove safety when other operations abstracted from the SMT encoding or when the counterexample feasibility checks were inefficient. We have dealt with these problems in Chap. 6 with a set of lattices and a counterexample feasibility check with non-linear arithmetic instead of propositional logic. I intend to explore this direction in-depth in future work.

We introduced LB-CEGAR, a full and generalised algorithm of the LB-CEGAR-BMC algorithm, in Chap. 6. The algorithm represented a function efficiently via a lattice of literals (as first-order formulas) and gradually refines the current representation of this function according to the partial order of literals. We have formalized LB-CEGAR such as that there was no more a requirement to have or use the full definition of a library function. We were able to remove this requirement by using a set of additional techniques and heuristics to deal efficiently with this scenario. The LB-CEGAR is a generalization of LB-CEGAR-BMC in that sense that the majority of these techniques and heuristics have not assume a specific program structure and in particular, have not require the code to be a loop-free program.

In the evaluation of LB-CEGAR on benchmarks with trigonometric, we observed that LB-CEGAR verified the highest number of **Safe** instances, on many on which other tools failed. The LB-CEGAR performed almost as well as HiFrog without any summary; in future work, I intend to examine additional techniques to achieve the same resource consumption as HiFrog with light-weight theories (Chap. 4).

We have introduced the idea of theory refinement. The new approach modularly adjusted the modelling precision based on the properties being verified through the use of a combination of theories. The approach verified a program via light-weight theories, and yet, it avoided spurious counterexample once part of the program (or its specification) was not expressible in a light-weight theory efficiently. The counterexample-guided theory refinement algorithm used the partial order according to precision between SMT theories to gradually refine (some of) the program statements to a bit-level precision. The algorithm guided by spurious counterexamples and the location of statements in a loop-free program identified and refined only the statements required for proving the correctness of the model via theory combination. We have formalised the notation for an efficient combination of two theories via a binding formula for the communication between two SMT theories. We demonstrated the process with two new theories called UFP and BVP.

We have described several heuristics with an evaluation of the effect of the refinement order of statements in the code on the resource consumption of the algorithm. When benchmarking the theory refinement algorithm, we observed that HiFrog with theory refinement was comparable to CBMC on safe instances; bug detection by HiFrog with theory refinement was slower than the one by CBMC. The problem was in the iterative refining of all the expressions to confirm the validity of the counterexample. I discuss how to handle the problem with smarter counterexample feasibility checks and improved heuristics in the future work section.

In the incremental verification approach followed in this thesis, the model checker verifies each property separately. The theory-aware summary refinement approach, a novel approach, used this idea to model a program using the lightest

possible theories to verify the desired property. That is, the approach lazily refined the encoding at the level of a property (in the LB-CEGAR-BMC, the LB-CEGAR, and theory refinement algorithms, Chap. 5, Chap. 6 and Chap. 7, respectively, the refinement has been done at the level of program statements).

The approach allowed using function summaries between different theories for incremental verification of software with many requirements via a theory interface mechanism. The conversion carried out between different SMT encodings of the same function summary was done via the theory interface. We have formalised the encoding and the decoding of summaries in a specific theory into and from the theory interface and presented the summary and theory-aware model checking algorithm that combined summary refinement with theory-aware refinement over an incremental verification framework using this mechanism. In the evaluation of the approach, we compared the approach with CBMC and HiFrog as in Chap. 4, and in overall, the results showed a significant positive impact on the effectiveness and efficiency of verification of large-scale and multi-property benchmarks of the theory-aware summary refinement approach.

## 9.2 Future Work

Following the evaluation discussions in Chap.5-Chap.8, the thesis conclusion and summary, I consider several research directions for the framework in general, the refinement techniques and possible combinations of the techniques. In the following sections, I describe the future work in more details in the SMT-based incremental model checking framework.

**Unify refinement mechanism in HiFrog.** I have formalised the high-level architecture of HiFrog for all refinement approaches presented in this thesis (see

Fig. 1.1 and Fig. 2.11 for more details). Since these refinement techniques are part of a single refinement flow (the *inner cycle* as described in Sec. 2.5), HiFrog can benefit from a combination of these refinement techniques when verifying a property. In Fig. 9.1, I present a benchmark with both bit-vectors operations and mathematical functions. This benchmark can be solved efficiently with a combination of theory refinement and LB-CEGAR.

```
1  int main()
2  {
3      /* Compute modulus division by (1 << s) - 1 without a division operator */
4      unsigned int n = __VERIFIER_nondet_uint(); /* numerator */
5      unsigned int s = __VERIFIER_nondet_uint();      /* s > 0 */
6      unsigned int d;
7      unsigned int m;                          /* n % d goes here.*/
8      __VERIFIER_assume(s < 32);
9      d = (1 << s) - 1; /* so d is either 1, 3, 7, 15, 31, ...)*/
10     if (d > 0) {
11         m = n;
12         while (n > d) {
13             m = 0;
14             while (n > 0) {
15                 m += n & d;
16                 n = n >> s;
17             }
18             n = m;
19         }
20         /* Now m is a value from 0 to d, but since with modulus division
21          * we want m to be 0 when it is d. */
22         if (m == d) {
23             m = 0;
24         }
25         assert(m == n % d);
26     }
27     return 0;
28 }
```

**Figure 9.1:** The benchmark modulus_false-no-overflow.c taken from SV-COMP (bit-vectors set). The benchmark combines bit operations with mathematical functions.

The first step I recognize in this research direction will be investigating the connection and the communication between pairs of refinement techniques, which we already have partially considered in Chap. 8 when using the refinement algorithm between theories refinement (Chap. 8) and summary refinement (Chap. 4). Once understanding better and formalising the communication between pairs of

refinement techniques, we can think and reason how to apply a general mechanism for allowing all the refinement techniques to communicate during verification of a property, efficiently.

**Theory refinement support for a combination of several theories.** In Chap. 7, we have described the combination of two theories via a binding formula and demonstrated the process with UFP and BVP. As a first step, we shall think and reason how to extend the binding formula support to other pairs of theories. Then, we shall consider and identify which verification problems can benefit from a theory combination between several theories in the context of bounded model checking. Last, we will try to extend the current communication via a binding formula to a general mechanism to communicate between several theories. We still intend the communication in our theory combination mechanism to be via a binding formula, but this formula will refer to the binding of equations of different theories between the theories, instead of two.

**Refinement heuristics for theory refinement.** In the evaluation of the theory refinement algorithm in Chap. 7, we examined the efficiency of 8 different heuristics in the context of BMC. The algorithm has performed well for loop-free programs. However, an evaluation with a random set of benchmarks with mathematical functions (modulo and elementary operations) in Chap. 5, yielded false results or an error. These were time-out, out-of-memory, and other related errors for more than half of the set. Theses results usually resulted from the solving phase itself when the set of heuristics was not designed to solve an instance. These errors could have been handled by adjusting the algorithm whenever being introduced with an unexpected new case. We believe that a more general solution should be applied.

We will investigate the reason for these errors and performance issues and will think and reason how to integrate new refinement heuristics. One of the problems we have already observed is related to the order of refinement when we started refining too early. We illustrate our early findings in a small crafted toy-example.

```
1  void main()
2  {
3    int a = 0;
4    while (1);
5    {
6      a = b % a;
7      b = abs(b) + 1;
8      c = abs(c) + 1;
9      d = ((c+b) % a) % d;
10     assert((a + b + c + d) > 1);
11   }
12 }
```

**Figure 9.2:** *Code Example: Refine too early from the assignment of a, when only b and c are needed.*

The spurious counterexample can be $\{a = -500, b = 0, c = 0, d = -1000\}$. The current heuristics in Chap. 7, would refine the assignment of $a$ or $d$, hence creating an expensive encoding (and gives nothing, since they are evaluated to 0 anyhow). For the assert statement in this example, it is actually enough to refine the assignments of $c$ and $b$ which are simple (and both $>= 1$). None of the heuristics presented in Chap. 7 could assist us in that case. That is the case when we wish to refine only several instructions in between the entry point and the assert.

We will consider computing real, correct executions to assist in finding a set of statements to refine regardless of their order in the code. This solution will work in general and not only in bounded model checking. We plan to try the following process:

1. Get a spurious counterexample from the abstract model.

2. Fix the counterexample using the bit-precise representation (that is, correct the values in the assignments in the counterexample itself).

3. Return a concrete counterexample or repeat (2) for different concrete values.

   Note that, we construct either a concrete counterexample or a concrete safe execution in the previous step. If we have a concrete counterexample, we will terminate the refinement and return this example.

In (2), we aim to find a minimal fix; however, we have not yet considered any strategy of technique.

Here, instead of examining each of the program statements locally, we will examine the assignment of each of the values in the counterexample and alter the counterexample accordingly. Only then, we will think and reason which of the statement to refine (not necessarily by the order of the statements in the loop-free program).

Given an amended counterexample or a set of amended counterexamples, we will consider the following *strategies of refinement*:

1. Fix all instructions that disagree.

2. Find the minimal set of All disagrees to fix.

3. Fix statistically all instructions that disagree.

4. Inner structure (e.g., a tree and fix all left siblings).

Each heuristic can be a combination of the heuristics presented in the evaluation of Chap. 7 and strategies 2-4. We will consider any of the combinations and examine the effect each of the new heuristics has on the performance and the ability to solve previously unsolved mathematical and algebraic benchmarks.

As the main contribution of this thesis is the lattice-based CEGAR approach, I describe the research direction related to the application and improvement of LB-CEGAR in more details.

**Efficient counterexample feasibility check for SMT-based model checking.** In Chap. 5, we have observed that the use of counterexample feasibility check in the experiments of LB-CEGAR-BMC had no positive effect of the performance in comparison to a version of LB-CEGAR-BMC with this check disabled. Moreover, in Chap. 6, we observed that LB-CEGAR outperformed the rest of the approaches (unlike LB-CEGAR-BMC in the experiments) partially due to implementation of lighter feasibility checks (via non-liner arithmetics instead of bit-vectors). It encourages us to explore this direction further.

We plan to add new heuristics to identify the most efficient theory or theory combination for a counterexample feasibility check (e.g., NRA for $sin()$ and NIA for % operation). In addition, we will examine satisfying assignments of false results from the solver to check the possibility to use linear arithmetics instead of non-linear arithmetics. With an interpretation for the variables or a partial interpretation of uninterpreted functions and uninterpreted predicates in the case of EUF, we believe counterexample feasibility check can be done more efficiently for some of the library functions and operations we refine. However, we have not yet examined these traces to come to any conclusion. The theory refinement approach can also benefit from this study.

We view the programs with trigonometric functions as the primary domain of application of LB-CEGAR; however, the standard decision problem in SMT is undecidable for formulas over the Reals with transcendental functions. Therefore, we consider removing the counterexample feasibility check with NRA and

using instead SMT solver to decide $\delta$-satisfiability or unsatisfiability formulas for the counterexample feasibility checks. Note that, we will still use SMT solvers to determine the satisfiability of formula during the refinement. The $\delta$-decision problem decides whether a formula is unsatisfiable or $\delta$-satisfiable ($\delta$ is a positive rational number). We will have to evaluate and examine the effect on the false result of LB-CEGAR (the false-SAT rate) with $\delta$-SAT as we describe here.

**Additional heuristics for LB-CEGAR.** The incremental solving mode in LB-CEGAR implementation was used merely for improving the resource consumption of the algorithm. After refining a spurious counterexample, we only added a subset of guarded literals to the query between two different checks for satisfiability. Since checking the satisfiability of a problem is more expensive (in time and memory) than checking the feasibility of a counterexample, we can benefit from controlling the number of guarded literals added after introduced with spurious counterexamples. Currently, we added the smallest subset of guarded literals required for refining a spurious counterexample.

We will try to find the right balance between the size of the query and the number of feasibility checks of a counterexample between two different checks for satisfiability, to avoid overly large queries and refinement of the same location in the code over and over. Note that, large queries will at some point suffer from the same problems that we observed in the evaluation of LB-CEGAR-BMC when using UDS (or flat lattices), which we intend to avoid at all costs. On the other hand, each cycle of LB-CEGAR does consume resources regardless of the size of the subset of guarded literals recently added to the query.

We will learn the effect on the performance of LB-CEGAR by controlling the following parameters:

1. The number of counterexamples the solver produces each call to satisfiability check of the problem, which is a positive integer $x$.

2. The stopping condition[1] of $traverse_{SAT}$ method (e.g., traversing to upper elements at most $k$ times or until finding a subset of guarded literals which refines $y$ counterexamples out of $x$).

3. The number of variables (or expressions in its SMT summary) in the problem which will be used in the interpretation of non-deterministic expressions in the encoding of the guarded literals.

We also expect these parameters to affect the false results rate of LB-CEGAR (in a positive way). Mainly, if each partition (the set of equation pushed to the solver between two checks for satisfiability) will contain a subset of guarded literals of different elements (of different lattices) when expressing dependence between these elements is essential for solving the current verification problem. Thus the results of this direction extend beyond performance improvements.

Last, we also consider the integration of LB-CEGAR with other solving approaches (in addition to SMT solving) for robotic benchmarks. However, we have not yet examined a specific solving approach.

**Over-approximate models via frontiers of satisfiability.** At the end of a successful refinement with LB-CEGAR, we construct a frontier of satisfiability for all occurrences of library functions in the code (assuming each of the library functions have a lattice). We can use these frontiers for

1. Constructing a set of assumptions under which this program is safe.

[1]Currently, it is when the current counterexample was refined or when no guarded literal can refine the current counterexample.

2. Creating an over-approximate model of the actual implementation of these library functions.

3. Extracting of function summary per occurrence of a library function in incremental verification approach (instead of using interpolation, as this set, at that point, is given to as with no additional cost).

4. Given two programs $X$ and $Y$, which are related or have some algorithmic similarity, we can use the frontier of satisfiability of the former to speed-up the verification process of the latter. That is, once we have verified program $X$, we will use the frontier from $X$ to verify $Y$. For example, this technique can be useful in verification of real-time systems, security code with the same basic functionality, or inner organisation code which shares the same utilities with mathematical functions.

We would like to extend the discussion here regarding creating an over-approximate model of the implementation of a library function.

Trigonometric and mathematical functions are common in control code for geometric calculations (e.g., ROS and drone control). In order to verify such code, we need a model or implementation of these functions. However, the relevant standards (ISO-9899 [ERM97], IEEE-754 [MB16], etc.) give only a very loose specification, such as special values evaluation (e.g., trigonometric tables as a specification).

The actual implementations of these functions may vary in terms of accuracy and will not be necessarily easy or possible to describe with simple parameters such as error bounds. Moreover, sometimes the implementations of these functions are not available (for example, these are depending on the compiler flag, x86 architecture, the SSE implementation or a software implementation), mainly

since the documentation of hardware and firmware implementation may not be complete, clear, useful or even correct. However, even when implementations are available, using them directly may cause significant performance overhead.

Finally, even if the exact implementation is available and tractable, it may not be desirable to verify code using it. In the case of *cyber-physical* systems, the error will likely swamp non-exceptional numerical errors. Especially when a control system relies on computing trigonometric functions perfectly and accurately, this will raise additional concerns regarding the behaviour of implementations and in the real world.

We plan to use lattices of guarded literals for transcendental functions (ordered from the most stringent to the least stringent, which is likely to require a semi-manual ranking for the transcendental functions). By using a frontier of satisfiability, we will think and reason how to identify the least restrictive set of guarded literals[2] required to verify the desired specification of a system. We will extract sets of frontiers by examples; we will consider the SMT-theories of the reals, integers and floating points ($QF\_FP$). We expect the method to be useful for verifying a software or a system (that is, for checking the actual implementations of these functions used in a specific architecture).

## 9.3 List of Publications

1. Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even-Mendoza, Grigory Fedyukovich, Antti E. J. Hyvärinen, Natasha Sharygina. HiFrog: SMT-based Function Summarization for Software Verification. *Tools and Algo-*

[2]This set of guarded literals is equivalent to some set of properties of a transcendental function.

*rithms for the Construction and Analysis of Systems (TACAS (2) 2017)*: 207-213, 2017

2. Antti E. J. Hyvärinen, Sepideh Asadi, Karine Even-Mendoza, Grigory Fedyukovich, Hana Chockler, Natasha Sharygina. Theory Refinement for Program Verification. *International Conference on Theory and Applications of Satisfiability Testing (SAT 2017)*: 347-363, 2017

3. Karine Even-Mendoza, Sepideh Asadi, Antti E. J. Hyvärinen, Hana Chockler, Natasha Sharygina. Lattice-Based Refinement in Bounded Model Checking. *Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2018)*: 50-68, 2018

4. Sepideh Asadi, Martin Blicha, Grigory Fedyukovich, Antti E. J. Hyvärinen, Karine Even-Mendoza, Natasha Sharygina, Hana Chockler. Function Summarization Modulo Theories. *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2018)*: 56-75, 2018

5. Karine Even-Mendoza, Antti E. J. Hyvärinen, Hana Chockler, Natasha Sharygina. Lattice-based SMT for program verification. *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE 2019)*: 16:1-16:11, 2019

The first publication's order of authors is alphabetic as each author's contribution was significant. In the rest of the publications, the first author is the lead author and primarily responsible for the paper, while Hana Chockler and Natasha Sharygina are both supervising the research.

# Bibliography

[AAB+07] Rajeev Alur, Marcelo Arenas, Pablo Barcelo, Kousha Etessami, Neil Immerman, and Leonid Libkin. First-order and temporal logics for nested words. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 151–160. IEEE, July 2007.

[AAC+17] Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even Mendoza, Grigory Fedyukovich, Antti E J Hyvärinen, and Natasha Sharygina. HiFrog: SMT-based function summarization for software verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10206 of *LNCS*, pages 207–213. Springer, 2017.

[ABE+05] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, Mihalis Yannakakis, and Mihalis Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):786–818, 2005.

[ABE18] Rajeev Alur, Ahmed Bouajjani, and Javier Esparza. *Model Checking Procedural Programs*, pages 541–572. Springer, 2018.

[ABG+12] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Lazy abstraction with interpolants

for arrays. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *LNCS*, pages 46–61. Springer, 2012.

[AFHS15] Leonardo Alt, Grigory Fedyukovich, Antti E J Hyvärinen, and Natasha Sharygina. A proof-sensitive approach for small propositional interpolants. In *Verified Software: Theories, Tools, and Experiments*, volume 9593 of *LNCS*, pages 1–18. Springer, 2015.

[AGC12] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *LNCS*, pages 39–55. Springer, 2012.

[AGT08] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, page 367–381. Springer, 2008.

[AHAS17] Leonardo Alt, Antti Eero Johannes Hyvärinen, Sepideh Asadi, and Natasha Sharygina. Duality-based interpolation for quantifier-free equalities and uninterpreted functions. In *Formal Methods in Computer-Aided Design*, FMCAD '17, pages 39–46. IEEE, 2017.

[AHS17] Leonardo Alt, Antti E J Hyvärinen, and Natasha Sharygina. LRA interpolants from no man's land. In *Hardware and Software: Verification and Testing*, volume 10629 of *LNCS*, pages 195–210. Springer, 2017.

[AIKY95] Rajeev Alur, Alon Itai, Robert P Kurshan, and Mihalis Yannakakis.

Timing verification by successive approximation. *Information and Computation*, 118(1):142–157, 1995.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.

[All07] Julia Allen. Why is security a software issue? *The EDP Audit, Control, and Security Newsletter (EDPACS)*, 36(1):1–13, 2007.

[ALS08] Zaher S Andraus, Mark H Liffiton, and Karem A Sakallah. Reveal: a formal verification tool for Verilog designs. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *LNCS*, pages 343–352. Springer, 2008.

[AM13] Aws Albarghouthi and Kenneth L McMillan. Beautiful interpolants. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 313–329. Springer, 2013.

[AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *International SPIN Workshop on Model Checking of Software, SPIN 2006*, volume 3925 of *LNCS*, pages 146–162. Springer, 2006.

[AMP09] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, Feb 2009.

[And87] Ian Anderson. *Combinatorics of Finite Sets*. Clarendon Press, Oxford, 1987.

[And11] Marc Andreessen. Why software is eating the world. [online] WSJ. https://www.wsj.com/articles/SB10001424053111903480904576512250915629460, 2011.

[AP10] Behzad Akbarpour and Lawrence Charles Paulson. MetiTarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.

[AQR+04] Tony Andrews, Shaz Qadeer, Sriram K Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Computer Aided Verification*, pages 484–487. Springer, 2004.

[Arm09] Alessandro Armando. Building SMT-based software model checkers: An experience report. In *Frontiers of Combining Systems*, volume 5749 of *LNCS*, pages 1–17. Springer, 2009.

[AV14] Waldo A F Alencar and Emilia Villani. Model checking applied to embedded software of university satellite. *Journal of Control, Automation and Electrical Systems*, 25(1):126–136, Feb 2014.

[AY01] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):273–303, 2001.

[BAM06] Sebastian Burckhardt, Rajeev Alur, and Milo M K Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 489–502. Springer, 2006.

[BB09] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:165–201, 2009.

[BBB+10] Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Caillaud, Benoît Delahaye, and Axel Legay. Statistical abstraction and model-checking of large heterogeneous systems. FMOODS 2010. In *Formal Techniques for Distributed Systems*, volume 6117 of *LNCS*, pages 32–46. Springer, 2010.

[BBC+06a] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review (OSR)*, 40(4):73–85, April 2006.

[BBČ+06b] Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Ročkai, and Pavel Šimeček. DiVinE – a tool for distributed verification. In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006.

[BBDE+97] Ilan Beer, Shoham Ben-David, Cindy Eisner, Daniel Geist, Leonid Gluhovsky, Tamir Heyman, Avner Landver, P Paanah, Yoav Rodeh, G Ronin, and Yaron Wolfsthal. RuleBase: Model checking at IBM. In *Computer Aided Verification*, volume 1254 of *LNCS*, pages 480–483. Springer, 1997.

[BBEL96] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. RuleBase: an industry-oriented formal verification tool. In *33rd De-*

*sign Automation Conference Proceedings, DAC 1996*, pages 655–660. IEEE, June 1996.

[BBH+13] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0 – an explicit-state model checker for multithreaded C & C++ programs. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.

[BBS11] Bryan A Brady, Randal E Bryant, and Sanjit A Seshia. Learning conditional abstractions. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 116–124. FMCAD Inc., 2011.

[BCC+99] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings 1999 Design Automation Conference, DAC 1999*, pages 317–320. IEEE, 1999.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Ce-

sare Tinelli. CVC4. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[BCF+07] Roberto Bruttomesso, Alessandro Cimatti, Andres Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems. In *Computer Aided Verification*, volume 4590 of *LNCS*, pages 547 – 560. Springer, 2007.

[BCM+92] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and computation*, 98(2):142–170, 1992.

[BCMD90] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, and David L Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 46–51. ACM, 1990.

[BD16] Dirk Beyer and Matthias Dangl. SMT-based software model checking: An experimental comparison of four algorithms. In *Verified Software: Theories, Tools, and Experiments*, volume 9971 of *LNCS*, pages 181–198. Springer, 2016.

[BDW15] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-induction with continuously-refined invariants. In *Computer Aided Verification*, volume 9206 of *LNCS*, pages 622–640. Springer, 2015.

[BEG+07] Sharon Barner, Cindy Eisner, Ziv Glazberg, Daniel Kroening, and Ishai Rabinovitz. ExpliSAT: Guiding SAT-based software verification

with explicit states. In *Hardware and Software: Verification and Testing*, volume 4383 of *LNCS*, pages 138–154. Springer, 2007.

[BG18] Dmitry Brizhinev and Rajeev Goré. A case study in formal verification of a Java program. *arXiv preprint arXiv:1809.03162*, 2018.

[BGP99] Tevfik Bultan, Richard Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.

[BH08] Domagoj Babic and Alan J Hu. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 211–220. ACM, 2008.

[BHJM05] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with Blast. In *Fundamental Approaches to Software Engineering*, volume 3442 of *LNCS*, pages 2–18. Springer, 2005.

[BHJM07] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, Oct 2007.

[BHMV05] Ahmed Bouajjani, Peter Habermehl, Pierre Moro, and Tomáš Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 13–29. Springer, 2005.

[Bir67] Garrett Birkhoff. *Lattice Theory*. AMS, 3rd edition, 1967.

[BK11] Dirk Beyer and M Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.

[BK18] Armin Biere and Daniel Kröning. *SAT-Based Model Checking*, pages 277–303. Springer, 2018.

[BKW07] Gérard Basler, Daniel Kroening, and Georg Weissenbacher. SAT-based summarization for Boolean programs. In *Model Checking Software. SPIN 2007*, volume 4595 of *LNCS*, pages 131–148. Springer, 2007.

[BLN03] Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A tool for BDD-based verification of real-time systems. In *Computer Aided Verification*, volume 2725 of *LNCS*, pages 122–125. Springer, 2003.

[BLN+13] Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. Precision reuse for efficient regression verification. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 389–399. ACM, 2013.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[Boc82] Gregor V Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, 31(3):223–231, March 1982.

[Boe19] 737 MAX software updates. https://www.boeing.com/commercial/737max/737-max-software-updates.page, Date Accessed September 05, 2019.

[Bon16] David Bonvoisin. 25 years of formal methods at ratp. *International Railway Safety Council (IRSC)*, 2016.

[Bou92] Raymond T Boute. The euclidean definition of the functions Div and Mod. *ACM Transactions on Programming Languages and Systems*, 14(2):127–144, April 1992.

[BPST10] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *LNCS*, pages 150–153. Springer, 2010.

[BR00] Thomas Ball and Sriram K Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 113–130. Springer, 2000.

[BR02a] Thomas Ball and Sriram K Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 1–3. ACM, 2002.

[BR02b] Tom Ball and Sriram K Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical report, Technical Report MSR-TR-2002-09, Microsoft Research, January 2002.

[Bra11] Aaron R Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[Bry86] Randal Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard version 2.0, 2010.

[BT19] Clark Barrett and Cesare Tinelli. Towards bit-width-independent proofs in SMT solvers. In *Automated Deduction–CADE 27: 27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings*, volume 11716 of *LNAI*, page 366. Springer, 2019.

[BVWW09] Thomas Bochot, Pierre Virelizier, Hélene Waeselynck, and Virginie Wiels. Model checking flight control systems: The Airbus experience. In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 18–27. IEEE, May 2009.

[BW12] Dirk Beyer and Philipp Wendler. Algorithms for software model checking: Predicate abstraction vs. impact. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 106–113. IEEE, Oct. 2012.

[cbm19] `http://www.cprover.org/cbmc/`, Date Accessed May 01, 2019.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM*

*SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252. ACM, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282. ACM, 1979.

[CC97] Sérgio Campos and Edmund Clarke. The verus language: Representing time efficiently with BDDs. In *Transformation-Based Reactive Systems Development*, volume 1231 of *LNCS*, pages 64–78. Springer, 1997.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[CCG+04] Sagar Chaki, Edmund M Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.

[CCK+02] Pankaj Chauhan, Edmund Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Formal Methods in Computer-Aided Design*, volume 2517 of *LNCS*, pages 33–51. Springer, 2002.

[CCM12] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. Theories,

solvers and static analysis by abstract interpretation. *Journal of the ACM*, 59(6):31:1–31:56, 2012.

[CdLF16] Lucas C Cordeiro and Eddie Batista de Lima Filho. SMT-based context-bounded model checking for embedded systems: Challenges and future trends. *ACM SIGSOFT Software Engineering Notes*, 41(3):1–6, 2016.

[CE82] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.

[CES86] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, apr 1986.

[CFMS12] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38(4):957–974, July 2012.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35. ACM, 1989.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form

and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CG18] Sagar Chaki and Arie Gurfinkel. *BDD-Based Symbolic Model Checking*, pages 219–245. Springer, 2018.

[CGH+93] Edmund M Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E Long, Kenneth L McMillan, and Linda A Ness. Verification of the futurebus+ cache coherence protocol. In *Proceedings of the 11th IFIP WG10.2 International Conference Sponsored by IFIP WG10.2 and in Cooperation with IEEE COMPSOC on Computer Hardware Description Languages and Their Applications*, CHDL '93, pages 15–30. Elsevier, 1993.

[CGH+15] Zhe Chen, Yi Gu, Zhiqiu Huang, Jun Zheng, Chang Liu, and Ziyi Liu. Model checking aircraft controller software: a case study. *Software: Practice and Experience*, 45(7):989–1017, 2015.

[CGI+17a] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10205 of *LNCS*, pages 58–75. Springer, 2017.

[CGI+17b] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Satisfiability modulo transcendental functions via incremental linearization. In *Automated Deduction – CADE 26*, volume 10395 of *LNCS*, pages 95–113. Springer, 2017.

[CGI+18a] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and

Roberto Sebastiani. Experimenting on solving nonlinear integer arithmetic with incremental linearization. In *Theory and Applications of Satisfiability Testing – SAT 2018*, volume 10929 of *LNCS*, pages 383–398. Springer, 2018.

[CGI+18b] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. *ACM Transactions on Computational Logic*, 19(3):19:1–19:52, 2018.

[CGJ+00] Edmund M Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[CGJ+01] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. *Progress on the State Explosion Problem in Model Checking*, volume 2000 of *LNCS*, pages 176–194. Springer, 2001.

[CGJ+03] Edmund M Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGL94] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGM05] Witold Charatonik, Lilia Georgieva, and Patrick Maier. Bounded model checking of pointer programs. In *Computer Science Logic. CSL 2005*, volume 3634 of *LNCS*, pages 397–412. Springer, 2005.

[CGP99] Edmund M Clarke, Jr., Orna Grumberg, and Doron A Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[CGP+02] Alessandro Cimatti, Enrico Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Integrating BDD-based and SAT-based symbolic model checking. In *Frontiers of Combining Systems*, volume 2309 of *LNCS*, pages 49–56. Springer, 2002.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.

[CIY95] Rance Cleaveland, Purush Iyer, and Daniel Yankelevich. Optimality in abstractions of model checking. In *Static Analysis*, volume 983 of *LNCS*, pages 51–63. Springer, 1995.

[CKK+18] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. JBMC: A bounded model checking tool for verifying Java bytecode. In *Computer Aided Verification*, volume 10981 of *LNCS*, pages 183–190. Springer, 2018.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[CKNZ12] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, volume 7682 of *LNCS*, pages 1–30. Springer, 2012.

[CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.

[Cla97] Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, pages 54–56. Springer, 1997.

[CLM91] Edmund M Clarke, David E Long, and Kenneth L McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, 79(9):1283–1292, Sep. 1991.

[CLM+10] Gianpiero Cabodi, Luciano Lavagno, Marco Murciano, Alex Kondratyev, and Yosinori Watanabe. Speeding-up heuristic allocation, scheduling and binding with SAT-based abstraction/refinement techniques. *ACM Transactions on Design Automation of Electronic System*, 15(2), March 2010.

[CMNQ06] Gianpiero Cabodi, Marco Murciano, Sergio Nocco, and Stefano Quer. Stepping forward with interpolants in unbounded model checking. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '06, pages 772–778. ACM, 2006.

[CNR13] Alessandro Cimatti, Iman Narasamdya, and Marco Roveri. Software model checking SystemC. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):774–787, May 2013.

[Com16] Competition on software verification (SV-COMP). `https://sv-comp.sosy-lab.org/2016/`, 2016.

[Com18] Competition on software verification (SV-COMP). `https://sv-comp.sosy-lab.org/2018/`, 2018.

[Com19] Competition on software verification (SV-COMP). `https://sv-comp.sosy-lab.org/2019/`, 2019.

[Cou00] Patrick Cousot. Partial completeness of abstract fixpoint checking. In *Abstraction, Reformulation, and Approximation*, volume 1864 of *LNCS*, pages 1–25. Springer, 2000.

[CPA19] `http://cpachecker.sosy-lab.org`, Date Accessed December 17, 2019.

[CPP14] Gianpiero Cabodi, Marco Palena, and Paolo Pasini. Interpolation with guided refinement: Revisiting incrementality in SAT-based unbounded model checking. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 12:43–12:50. IEEE, 2014.

[CPR13] Hana Chockler, Dmitry Pidan, and Sitvanit Ruah. Improving representative computation in ExpliSAT. In *Hardware and Software: Verification and Testing*, volume 8244 of *LNCS*, pages 359–364. Springer, 2013.

[cpr19] `http://www.cprover.org/`, Date Accessed May 01, 2019.

[Cra57] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[CU98] Michael A. Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, volume 1427 of *LNCS*, pages 293–304. Springer, 1998.

[DA02] Alexandre David and Tobias Amnell. Uppaal2k: small tutorial. *Available on-line at http://www.it.uu.se/research/group/darts/uppaal/tutorial.ps (September 7, 2007)*, 2002.

[DDLM11] Florent De Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2011.

[DdM06] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[DG18] Dennis Dams and Orna Grumberg. *Abstraction and Abstraction Refinement*, pages 385–419. Springer, 2018.

[DGG93] Dennis Dams, Orna Grumberg, and Rob Gerth. Generation of reduced models for checking fragments of CTL. In *Computer Aided Verification*, volume 697 of *LNCS*, pages 479–490. Springer, 1993.

[DHK14] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract satisfaction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 139–150. ACM, 2014.

[DHKR11] Alastair F Donaldson, Leopold Haller, Daniel Kroening, and Philipp

Rümmer. Software verification using k-Induction. In *Static Analysis*, volume 6887 of *LNCS*, pages 351–368. Springer, 2011.

[DKPW10] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *LNCS*, pages 129–145. Springer, 2010.

[DKW08] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[DLM09] Marc Daumas, David Lester, and César Muñoz. Verified real number calculations: A library for interval arithmetic. *IEEE Transactions on Computers*, 58(2):226–237, 2009.

[DM14] William Denman and César Muñoz. Automated real proving in pvs via MetiTarski. In *FM*, volume 8442 of *LNCS*, pages 194–199. Springer, 2014.

[dMB08a] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, 2008. Part of special issue: Proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007).

[DMB08b] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[DMP13] Leonardo De Moura and Grant Olney Passmore. Computation in real closed infinitesimal and transcendental extensions of the rationals. In *Automated Deduction – CADE-24*, volume 7898 of *LNCS*, pages 178–192. Springer, 2013.

[DNS05] David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[EAH+18] Karine Even-Mendoza, Sepideh Asadi, Antti E J Hyvärinen, Hana Chockler, and Natasha Sharygina. Lattice-based refinement in bounded model checking. In *Verified Software: Theories, Tools, and Experiments*, volume 11294 of *LNCS*, pages 50–68. Springer, 2018.

[EC80] E Allen Emerson and Edmund M Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, volume 85 of *LNCS*, pages 169–181. Springer, 1980.

[EKS06] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 489–503. Springer, 2006.

[ELLL04] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5(2-3):247–267, 2004.

[EMA10] Niklas Een, Alan Mishchenko, and Nina Amla. A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, page 181–188. FMCAD Inc, 2010.

[EMB11] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, page 125–134. FMCAD Inc, 2011.

[EMT+17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *Computer Aided Verification*, volume 10427 of *LNCS*, pages 126–133. Springer, 2017.

[ERM97] P D Edwards, R S Rivett, and G F McCall. Towards an automotive 'safer subset' of C. In *Safe Comp 97*, pages 185–196. Springer, 1997.

[ES01] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *Computer Aided Verification*, volume 2102 of *LNCS*, pages 324–336. Springer, 2001.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing – SAT 2004*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[FB18] Grigory Fedyukovich and Rastislav Bodík. Accelerating syntax-guided invariant synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10805 of *LNCS*, pages 251–269. Springer, 2018.

[FBZ+18] Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Pierre-Emmanuel Hladik, Félix Ingrand, and Anthony Mallet. Formal verification of complex robotic systems on resource-constrained platforms. In *Proceedings of the 6th Conference on Formal Methods in Software Engineering*, FormaliSE '18, pages 2–9. ACM, 2018.

[FCHS15] Grigory Fedyukovich, Andrea Callia D'Iddio, Antti Eero Johannes Hyvärinen, and Natasha Sharygina. Symbolic detection of assertion dependencies for bounded model checking. In *FASE*, volume 9033 of *LNCS*, pages 186–201. Springer, 2015.

[FCN+10] Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. Applying SMT in symbolic execution of microcode. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, page 121–128. FMCAD Inc, 2010.

[FCSM93] José Luiz Fiadeiro, José Félix Costa, Amilcar Sernadas, and Tom S E Maibaum. Process semantics of temporal logic specification. In *Recent*

*Trends in Data Type Specification*, volume 655 of *LNCS*, pages 236–253. Springer, 1993.

[FHT+07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[FS15] Grigory Fedyukovich and Natasha Sharygina. Towards completeness in bounded model checking through automatic recursion depth detection. In *Formal Methods: Foundations and Applications*, volume 8941 of *LNCS*, pages 96–112. Springer, 2015.

[FSS13] Grigory Fedyukovich, Ondrej Sery, and Natasha Sharygina. eVolCheck: Incremental upgrade checker for C. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 292–307. Springer, 2013.

[FSS17] Grigory Fedyukovich, Ondrej Sery, and Natasha Sharygina. Flexible SAT-based framework for incremental bounded upgrade checking. *International Journal on Software Tools for Technology Transfer*, 19(5):517–534, 2017.

[Gar15] Vijay K Garg. *Introduction to Lattice Theory with Computer Science Applications*. Wiley Publishing, 1st edition, 2015.

[GBM14] Arie Gurfinkel, Anton Belov, and João Marques-Silva. Synthesizing safe bit-precise invariants. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 93–108. Springer, 2014.

[GBW+18] Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani. The JKind model checker. In *Computer Aided Verification*, volume 10982 of *LNCS*, pages 20–27. Springer, 2018.

[GDH14] Xiang Gan, Jori Dubrovin, and Keijo Heljanko. A symbolic model checking approach to verifying satellite onboard software. *Science of Computer Programming*, 82:44 – 55, 2014. Special Issue on Automated Verification of Critical Systems (AVoCS'11).

[GG06] Malay K Ganai and Aarti Gupta. Accelerating high-level bounded model checking. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 794–801. ACM, 2006.

[GH02] Peter R Gluck and Gerard J Holzmann. Using SPIN model checking for flight software verification. In *Proceedings, IEEE Aerospace Conference*, volume 1, page 105–113. IEEE, March 2002.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *Computer Aided Verification*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[Git19a] Git repository of evaluation of lattice-based refinement approach. `https://github.com/karineek/latticeref`, 2019.

[Git19b] Git repository of HiFrog. `https://scm.ti-edu.ch/projects/hifrog/`, Date Accessed May 07, 2019.

[GJBF18] Ning Ge, Eric Jenn, Nicolas Breton, and Yoann Fonteneau. Integrated formal verification of safety-critical software. *International Journal on Software Tools for Technology Transfer*, 20(4):423–440, 2018.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dreal: An SMT solver for nonlinear theories over the reals. In *Automated Deduction – CADE-24*, volume 7898 of *LNAI*, pages 208–214. Springer, 2013.

[GKKN15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The SeaHorn verification framework. In *Computer Aided Verification*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015.

[GKQT18] Aarti Gupta, Vineet Kahlon, Shaz Qadeer, and Tayssir Touili. *Model Checking Concurrent Programs*, pages 573–611. Springer, 2018.

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 174–186. ACM, ACM, 1997.

[God07] Patrice Godefroid. Compositional dynamic test generation. *ACM SIGPLAN Notices*, 42(1):47–54, January 2007.

[got19] `http://www.cprover.org/goto-cc/`, Date Accessed December 19, 2019.

[GQ01] Roberto Giacobazzi and Elisa Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. In *Static Analysis*, volume 2126 of *LNCS*, pages 356–373. Springer, 2001.

[GRS00] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[GS05] Anubhav Gupta and Ofer Strichman. Abstraction refinement for bounded model checking. In *Computer Aided Verification*, volume 3576 of *LNCS*, pages 112–124. Springer, 2005.

[GW93] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, Apr 1993.

[HAE+17] Antti E J Hyvärinen, Sepideh Asadi, Karine Even-Mendoza, Grigory Fedyukovich, Hana Chockler, and Natasha Sharygina. Theory refinement for program verification. In *Theory and Applications of Satisfiability Testing – SAT 2017*, volume 10491 of *LNCS*, pages 347–363. Springer, 2017.

[Hal13] Leopold Haller. *Abstract Satisfaction*. PhD thesis, University of Oxford, 2013.

[Har00] John Harrison. Formal verification of floating point trigonometric functions. In *Formal Methods in Computer-Aided Design, FMCAD 2000*, volume 1954 of *LNCS*, pages 217–233. Springer, 2000.

[HBJ+14] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barret, and Cesare Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *Computer Aided Verification*, volume 8559 of *LNCS*, pages 680 – 695. Springer, 2014.

[HCD+18] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate automizer and the search for perfect interpolants - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10806 of *LNCS*, pages 447–451, 2018.

[HCR+16] Yen-Sheng Ho, Pankaj Chauhan, Pritam Roy, Alan Mishchenko, and Robert Brayton. Efficient uninterpreted function abstraction and refinement for word-level model checking. In *Formal Methods in Computer-Aided Design*, FMCAD '16, pages 65–72. ACM, 2016.

[HHP10] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 471–482. ACM, 2010.

[HiF17a] `http://verify.inf.usi.ch/hifrog/theoref`, 2017.

[HiF17b] HiFrog - tool usage page. `http://verify.inf.usi.ch/hifrog/tool-usage`, 2017.

[HiF19] `http://verify.inf.usi.ch/content/lattice-refinement`, 2019.

[HL91] Gerard J Holzmann and William Slattery Lieberman. *Design and validation of computer protocols*, volume 512. Prentice hall Englewood Cliffs, 1991.

[HM19] Ákos Hajdu and Zoltán Micskei. Efficient strategies for CEGAR-based model checking. *Journal of Automated Reasoning*, pages 1–41, Nov 2019.

[HMAS16] Antti E J Hyvärinen, Matteo Marescotti, Leonardo Alt, and Natasha Sharygina. OpenSMT2: An SMT solver for multi-core and cloud computing. In *Theory and Applications of Satisfiability Testing – SAT 2016*, volume 9710 of *LNCS*, pages 547–553. Springer, 2016.

[Hoa71] Charles Antony Richard Hoare. Procedures and parameters: An axiomatic approach. *Symposium on Semantics of Algorithmic Languages*, pages 102–116, 1971.

[Hol97] Gerard J Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering - Special issue on formal methods in software practice*, 23(5):279–295, May 1997.

[Hol11] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.

[Hoo93] John N Hooker. Solving the incremental satisfiability problem. *The Journal of Logic Programming*, 15(1-2):177–186, 1993.

[HR17] Shaobo He and Zvonimir Rakamarić. Counterexample-guided bit-precision selection. In *Programming Languages and Systems*, volume 10695 of *LNCS*, pages 534–553. Springer, 2017.

[Hua95] Guoxiang Huang. Constructing Craig interpolation formulas. In *Computing and Combinatorics*, volume 959 of *LNCS*, pages 181–190. Springer, 1995.

[Hua15] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 165–174. ACM, 2015.

[HWZ08] Holger Hermanns, Björn Wachter, and Lijun Zhang. Probabilistic CEGAR. In *Computer Aided Verification*, volume 5123 of *LNCS*, pages 162–175. Springer, 2008.

[IGCS19] Ahmed Irfan, Alberto Griggio, Alessandro Cimatti, and Roberto Sebastiani. MathSAT5 (nonlinear) at the SMT competition 2019. *SMT Competition 2019*, 2019.

[Ios01] Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 254–261. IEEE, Nov 2001.

[IX18] Radu Iosif and Xiao Xu. Abstraction refinement for emptiness checking of alternating data automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10806 of *LNCS*, pages 93–111, 2018.

[IX19] Radu Iosif and Xiao Xu. Alternating automata modulo first order theories. In *Computer Aided Verification*, volume 11562 of *LNCS*, pages 43–63. Springer, 2019.

[IYG+05] F Ivančić, Zijiang Yang, Malay K Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-Soft: Software verification platform. In *Computer Aided Verification*, volume 3576 of *LNCS*, pages 301–306. Springer, 2005.

[JAF+16] Pavel Jančík, Leonardo Alt, Grigory Fedyukovich, Antti E J Hyvärinen, Jan Kofroň, and Natasha Sharygina. PVAIR: Partial variable

assignment InterpolatoR. In *Fundamental Approaches to Software Engineering*, volume 9633 of *LNCS*, pages 419–434. Springer, 2016.

[JM04] Michael Jones and Eric Mercer. Explicit state model checking with hopper. In *Model Checking Software. SPIN 2004*, volume 2989 of *LNCS*, pages 146–150. Springer, 2004.

[JM07] Ranjit Jhala and Kenneth L McMillan. Array abstractions from proofs. In *Computer Aided Verification*, volume 4590 of *LNCS*, pages 193–206. Springer, 2007.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, October 2009.

[JR11] Phillip James and Markus Roggenbach. Automatically verifying railway interlockings using SAT-based model checking. *Electronic Communications of the EASST*, 35, 2011.

[KBH15] Guy Katz, Clark Barrett, and David Harel. Theory-aided model checking of concurrent transition systems. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 81–88. IEEE, 2015.

[KdR85] Ron Koymans and Willem P de Roever. Examples of a real-time temporal logic specification. In *The Analysis of Concurrent Systems*, volume 207 of *LNCS*, pages 231–251. Springer, 1985.

[KGC16] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[KGP06] Raimund Kirner, Markus Grössing, and Peter P Puschner. Comparing WCET and resource demands of trigonometric functions implemented as iterative calculations vs. table-lookup. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, OASICS. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[KGTW12] Temesghen Kahsai, Pierre-Loïc Garoche, Cesare Tinelli, and Mike Whalen. Incremental verification with mode variable invariants in state machines. In *NASA Formal Methods*, volume 7226 of *LNCS*, pages 388–402. Springer, 2012.

[KHK19] Yunho Kim, Shin Hong, and Moonzoo Kim. Target-driven compositional concolic testing with function summary refinement for effective bug detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 16–26. ACM, 2019.

[KIY16] Takuro Kutsuna, Yoshinao Ishii, and Akihiro Yamamoto. Abstraction and refinement of mathematical functions toward SMT-based test-case generation. *International Journal on Software Tools for Technology Transfer*, 18(1):109–120, 2016.

[KKNP09] Mark Kattenbelt, Marta Kwiatkowska, Gethin Norman, and David Parker. Abstraction refinement for probabilistic software. In *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 182–197. Springer, 2009.

[Kra97] Jan Krajíček. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *The Journal of Symbolic Logic*, 62(2):457–486, 1997.

[Kri59] Saul A Kripke. A completeness theorem in modal logic. *The journal of symbolic logic*, 24(1):1–14, 1959.

[KRSS16] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. JayHorn: A framework for verifying Java programs. In *Computer Aided Verification*, volume 9779 of *LNCS*, pages 352–358. Springer, 2016.

[KS96] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer, 2008.

[KSK15] Shrawan Kumar, Amitabha Sanyal, and Uday P Khedker. Value slice: A new slicing concept for scalable property checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *LNCS*, pages 101–115. Springer, 2015.

[KST+08] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M Wintersteiger. Loop summarization using abstract transformers. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis. ATVA '08*, volume 5311 of *LNCS*, page 111–125. Springer, 2008.

[KST+09] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M Wintersteiger. Loopfrog: A static analyzer for ANSI-C programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 668–670. IEEE, 2009.

[KSU11] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. *ACM SIGPLAN Notices*, 46(6):222–233, June 2011.

[KT14] Daniel Kroening and Michael Tautschnig. CBMC – C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 389–391. Springer, 2014.

[Kur94] Robert P Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, USA, 1994.

[KW11] Daniel Kroening and Georg Weissenbacher. Interpolation-based software verification with Wolverine. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 573–578. Springer, 2011.

[LGS+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, Saddek Bensalem, and David Probst. Property preserving abstractions for the verification of concurrent systems. *Formal methods in system design*, 6(1):11–44, 1995.

[lib19] libcurl - small example snippets. `https://curl.haxx.se/libcurl/c/example.html`, Date Accessed December 19, 2019.

[Lin06] Per Lindström. First-order logic. *Philosophical Communications, Web Series*, 36, 2006.

[Lis19] List of trigonometric identities, from Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/List_of_trigonometric_identities`, Date Accessed May 01, 2019.

[LMN15] Martin Leucker, Grigory Markin, and Martin R Neuhäußer. A new refinement strategy for CEGAR-based industrial model checking. In *Hardware and Software: Verification and Testing*, volume 9434 of *LNCS*, pages 155–170. Springer, 2015.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 97–107. ACM, 1985.

[LVB+12] Jussi Lahtinen, Janne Valkonen, Kim Björkman, J Frits, Ilkka Niemelä, and Keijo Heljanko. Model checking of safety-critical software in the nuclear engineering domain. *Reliability Engineering and System Safety*, 105:104–113, 2012. ESREL 2010.

[LW15] K Rustan M Leino and Valentin Wüstholz. Fine-grained caching of verification results. In *Computer Aided Verification*, volume 9206 of *LNCS*, pages 380–397. Springer, 2015.

[MB16] Monika Maan and Abhay Bindal. A review on IEEE-754 standard floating point arithmetic unit. *International Journal of Advance Research , Ideas and Innovations in Technology*, 2016.

[McM93a] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[McM93b] Kenneth L McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Kluwer Academic, 1993.

[McM02] Ken L McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 250–264. Springer, 2002.

[McM03] Kenneth L McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[McM05] Kenneth L McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[McM06] Kenneth L McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[McM10] Kenneth L McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification*, volume 6174 of *LNCS*, pages 104–118. Springer, 2010.

[McM14] Kenneth L McMillan. Lazy annotation revisited. In *Computer Aided Verification*, volume 8559 of *LNCS*, pages 243–259. Springer, 2014.

[Mel12] Guillaume Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, 2012. Special Issue: 8th Conference on Real Numbers and Computers.

[MHS18] Matteo Marescotti, Antti EJ Hyvärinen, and Natasha Sharygina. SMTS: Distributed, visualized constraint solving. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 57 of *EPiC Series in Computing*, pages 534–542. EasyChair, 2018.

[MM05] Guillaume Melquiond and Cesar Munoz. Guaranteed proofs using interval arithmetic. In *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pages 188–195. IEEE, 2005.

[MMN+12] Mikhail U Mandrykin, Vadim S Mutilin, Evgeny M Novikov, Alexey V Khoroshilov, and Pavel E Shved. Using linux device drivers for static verification tools benchmarking. *Programming and Computer Software*, 38(5):245–256, Sep 2012.

[MMZ+01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535. ACM, 2001.

[MO81] Yonatan Malachi and Susan S Owicki. *Temporal Specifications of Self-Timed Systems*, pages 203–212. Springer, 1981.

[Mod19] Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Modulo_operation`, Date Accessed May 01, 2019.

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation*, volume 9583 of *LNCS*, pages 41–62. Springer, 2016.

[Nes18] Michaela Nesvarova. 'software is everywhere'. [online] u-today (utoday.nl). https://www.utoday.nl/spotlight/64981/software-is-everywhere, 2018.

[NO79] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NO05] Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Computer Aided Verification*, volume 3576 of *LNCS*, pages 321–334. Springer, 2005.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.

[NRS14] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Ultimately incremental SAT. In *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *LNCS*, pages 206–218. Springer, 2014.

[OR11] C H Luke Ong and Steven J Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 587–598. ACM, 2011.

[osp19] Open Source Physics page. `http://www.compadre.org/osp/`, Date Accessed May 01, 2019.

[Pat19] Ron Patton. *Infamous Software Error Case Studies. In: Software testing.*, pages 10–13. Pearson Education India, 2019.

[PBG05] Mukul R Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, Apr 2005.

[Pel96] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, Jan 1996.

[Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *The Journal of Symbolic Logic*, 62(3):981–998, 1997.

[PW05] Andreas Podelski and Thomas Wies. Boolean heaps. In *Static Analysis*, volume 3672 of *LNCS*, pages 268–283. Springer, 2005.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[RAF+13] Simone Fulvio Rollini, Leonardo Alt, Grigory Fedyukovich, Antti E J Hyvärinen, and Natasha Sharygina. PeRIPLO: A framework for producing effective interpolants in SAT-based software verification. In

*Logic for Programming, Artificial Intelligence, and Reasoning*, volume 8312 of *LNCS*, pages 683–693. Springer, 2013.

[RDH03] Robby, Matthew B Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, page 267–276. ACM, 2003.

[RE14] Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *Computer Aided Verification*, volume 8559 of *LNCS*, pages 106–113. Springer, 2014.

[RG05] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *Computer Aided Verification*, volume 3576 of *LNCS*, pages 82–97. Springer, 2005.

[RHK13] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013.

[RHS95] Thomas W Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61. ACM, 1995.

[RICB17] Herbert Rocha, Hussama Ismail, Lucas Cordeiro, and Raimundo Barreto. *Model Checking Embedded C Software Using k-Induction and Invariants*, pages 159–182. Springer, 2017.

[RNO14] Steven J Ramsay, Robin P Neatherway, and C H Luke Ong. A type-directed abstraction refinement approach to higher-order model checking. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 61–72. ACM, 2014.

[ROS19] ROS homepage. `http://www.ros.org/`, Date Accessed May 01, 2019.

[RS13] Philipp Rummer and Pavle Subotic. Exploring interpolants. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '13, pages 69–76. IEEE, 2013.

[RTJB17] Andrew Reynolds, Cesare Tinelli, Dejan Jovanović, and Clark Barrett. Designing theory solvers with extensions. In *Frontiers of Combining Systems*, volume 10483 of *LNCS*, pages 22–40. Springer, 2017.

[Saï00] Hassen Saïdi. Model checking guided abstraction and analysis. In Jens Palsberg, editor, *Static Analysis*, volume 1824 of *LNCS*, pages 377–396. Springer, 2000.

[Sat19] Satisfiability modulo theories, solvers, from Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories#Solvers`, Aug 2019.

[SFB07] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 112–122. ACM, 2007.

[SFS12a] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. FunFrog: Bounded model checking with interpolation-based function summa-

rization. In *Automated Technology for Verification and Analysis*, volume 7561 of *LNCS*, pages 203–207. Springer, 2012.

[SFS12b] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-based function summaries in bounded model checking. In *Hardware and Software: Verification and Testing*, volume 7261 of *LNCS*, pages 160–175. Springer, 2012.

[Sht01] Ofer Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *Correct Hardware Design and Verification Methods*, volume 2144 of *LNCS*, pages 58–70. Springer, 2001.

[Sil12] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys (CSUR)*, 44(3):12:1–12:41, June 2012.

[SISG06] Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *Static Analysis*, volume 4134 of *LNCS*, pages 3–17. Springer, 2006.

[SKB+17] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Incremental bounded model checking for embedded software. *Formal Aspects of Computing*, 29(5):911–931, 2017.

[SS90] Gunnar Stålmarck and Mårten Säflund. Modeling and verifying systems and software in propositional logic. In *Safety of Computer Control Systems 1990 (Safecomp'90)*, pages 31–36. Elsevier, 1990.

[SS96] João P Marques Silva and Karem A Sakallah. GRASP—a new search algorithm for satisfiability. In *Proceedings of the 1996*

*IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227. IEEE, 1996.

[SS99] João P Marques Silva and Karem A Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[SS01] David PL Simons and Mariëlle IA Stoelinga. Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k. *International Journal on Software Tools for Technology Transfer*, 3(4):469–485, 2001.

[SSCS12] Nishant Sinha, Nimit Singhania, Satish Chandra, and Manu Sridharan. Alternate and learn: Finding witnesses without looking all over. In *Computer Aided Verification*, volume 7358 of *LNCS*, pages 599–615. Springer, 2012.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design*, FMCAD '00. Springer, 2000.

[Sum18] Summary and theory refinement [experimental]. `http://verify.inf.usi.ch/sum-theoref`, 2018.

[TB08] Sarah Thompson and Guillaume Brat. Verification of C++ flight software with the MCP model checker. In *2008 IEEE Aerospace Conference*, pages 1–9. IEEE, March 2008.

[TE14] Piotr Trojanek and Kerstin Eder. Verification and testing of mobile robot navigation algorithms: A case study in spark. In *2014*

*IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1489–1494. IEEE, 2014.

[The19a] The Coq proof assistant. `https://coq.inria.fr/`, Date Accessed May 01, 2019.

[The19b] The software-artifact infrastructure repository (SIR). `http://sir.unl.edu`, Date Accessed December 19, 2019.

[Top19] Gwyn Topham. Boeing 737 Max ordered by Ryanair undergoes name change. https://www.theguardian.com/business/2019/jul/15/boeing-737-max-ordered-by-ryanair-undergoes-name-change, 2019.

[Tri19] Trigonometric tables, from Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Trigonometric_tables`, Date Accessed May 01, 2019.

[TVKO17] Vu Xuan Tung, To Van Khanh, and Mizuhito Ogawa. raSAT: an SMT solver for polynomial constraints. *Formal Methods in System Design*, 51(3):462–499, Dec 2017.

[Var15] Mikko Varpiola. Software is everywhere. https://www.synopsys.com/blogs/software-security/software-is-everywhere/, 2015.

[VG09] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *2009 Formal Methods in Computer-Aided Design, FMCAD 2009*, pages 1–8. IEEE, 2009.

[VGM15] Yakir Vizel, Arie Gurfinkel, and Sharad Malik. Fast interpolating

BMC. In *Computer Aided Verification*, volume 9206 of *LNCS*, pages 641–657. Springer, 2015.

[vRAR11] Alexander von Rhein, Sven Apel, and Franco Raimondi. Introducing binary decision diagrams in the explicit-state verification of Java code. In *In the proceedings of Java Pathfinder Workshop*, page 82, 2011.

[VWM15] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, Nov 2015.

[VYY09] Martin Vechev, Eran Yahav, and Greta Yorsh. Experience with model checking linearizability. In *Model Checking Software. SPIN 2009*, volume 5578 of *LNCS*, pages 261–278. Springer, 2009.

[WBKW07] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent linux device drivers. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 501–504. ACM, 2007.

[WDF+15] Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Lee Koay, Kerstin Dautenhahn, and Joan Saez-Pons. Toward reliable autonomous robotic assistants through formal verification: a case study. *IEEE Transactions on Human-Machine Systems*, 46(2):186–196, 2015.

[WDH19] Shun Wang, Ye Du, and Zhen Han. An improved method of k-induction combined with predicate abstraction and CEGAR for software model checking. *Cluster Computing*, 22(3):6219–6229, May 2019.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE, 1981.

[Wie14] Siert Wieringa. *Incremental satisfiability solving and its applications*. PhD thesis, Aalto University; Aalto-yliopisto, 2014.

[Wit16] Jens Wittenburg. *Kinematics: Theory and Applications*. Springer, 2016.

[WKO13] Björn Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multi-threaded software with impact. In *Formal Methods in Computer-Aided Design*, pages 210–217. IEEE, Oct 2013.

[WKS01] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: a new incremental satisfiability engine. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, page 542–545. ACM, 2001.

[WSH+08] Bruce W Weide, Murali Sitaraman, Heather K Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In *Verified Software: Theories, Tools, and Experiments*, volume 5295 of *LNCS*, pages 84–98. Springer, 2008.

[XA05] Yichen Xie and Alexander Aiken. Saturn: A SAT-based tool for bug detection. In *Computer Aided Verification*, volume 3576 of *LNCS*, pages 139–143. Springer, 2005.

[Zha97] Hantao Zhang. SATO: An efficient propositional prover. In *International Conference on Automated Deduction—CADE-14*, volume 1249 of *LNCS*, pages 272–275. Springer, 1997.

[Zha03] Lintao Zhang. *Searching for truth: techniques for satisfiability of Boolean formulas*. PhD thesis, Princeton University Princeton, 2003.

[ZM02] Lintao Zhang and Sharad Malik. The quest for efficient Boolean satisfiability solvers. In *Computer Aided Verification*, volume 2404 of *LNCS*, pages 17–36. Springer, 2002.